.NET alapú programok minőségének és biztonságának növelése

Doktori értekezés

Pócza Krisztián

Témavezető:
Dr. Porkoláb Zoltán

Eötvös Loránd Tudományegyetem
Informatika Doktori Iskola
Az informatika alapjai és módszertana doktori program
Iskola- és programvezető: Dr. Demetrovics János

Budapest, 2010

Köszönetnyilvánítás

Elsőként témavezetőmnek, Dr. Porkoláb Zoltánnak szeretnék köszönetet mondani. Ő az, aki bevezetett a tudományok világába, amikor néhány éve megkezdtem doktori kutatásomat. Azóta folyamatosan támogatott, irányított, lelkiismeretesen mutatta az utat. Sok éves tapasztalatából mindent igyekezett átadni, amit csak lehetséges.

Köszönöm családomnak a támogatást, valamint azt, hogy olyan egyetemre járhattam, amely megalapozott tudást adott a doktori kutatásokhoz.

Köszönet illeti feleségemet, Veronikát, aki nélkülözött az alatt az idő alatt, amíg a cikkeimmel vagy éppen ezzel a dolgozattal foglalatoskodtam.

Szerzőtársaimnak köszönöm a közös munkát, amely a sikeres publikációk megjelenését tette lehetővé.

Végül, de nem utolsó sorban köszönöm a Programozási Nyelvek és Fordítóprogramok tanszék dolgozóinak, hogy mindig bizalommal fordulhattam hozzájuk.

Tartalomjegyzék

Ábrák jegyzéke

Táblázatok jegyzéke

Egyenletek jegyzéke

Bevezetés

Doktori kutatásom során azt a célt tűztem ki, hogy olyan módszereket hozzak létre valamint fejlesszek tovább, amelyek a .NET-programok minőségi és biztonsági jellemzőit javítják. Köztudott, hogy az elmúlt évtizedek során már számos ilyen módszer látott napvilágot, azonban ezek olyan környezetekre specializáltak, amelyek nehezen adaptálhatóak a .NET környezetre, illetve időközben technikailag túlhaladottá váltak.

A szabványos Microsoft .NET [99][81][85] technológia a 2002-es megjelenése óta dinamikus fejlődésen ment keresztül. Fontos megemlíteni, hogy mivel egy nagyon átgondolt és jól megtervezett keretrendszerről beszélünk, ezért a legelső verzióban napvilágot látott koncepciók még ma is megállják a helyüket. Mind a keretrendszer, mind pedig a nyelvi támogatás szempontjából a folyamatos és dinamikus bővülés jellemezte.

A .NET keretrendszer [14] már önmagában is segíti azt, hogy magas minőségi és biztonsági paraméterekkel rendelkező programokat állíthassunk elő. Ez annyit jelent, hogy olyan alapvető programozói hibák, amelyek pl. a puffer-túlcsordulásból, memóriakezelési anomáliákból, helytelen típuskezelésből, nem megbízható kód futtatásából adódnak, nehezebben vagy egyáltalán nem követhetők el. Azonban vannak olyan területek, amelyeket ez a rendszer sem fed le keretrendszer, alaprendszer mivoltából adódóan. Azt is figyelembe kell venni, hogy a konkrét programok (pl. elosztott alkalmazások) alkalmazásszintű biztonságának megőrzése, minőségének növelése még mindig az alkalmazás tervezőjének illetve kivitelezőjének a feladata. Ez az a terület, amely tudományos szempontból is rendelkezik kutatási lehetőségekkel.

Munkám során olyan módszereket dolgoztam ki, amelyek az önálló alkalmazások illetve az elosztott alkalmazások minőségi és biztonsági jellemzőit egyaránt javítják. A módszerek konkrét implementációit a .NET keretrendszerre készítettem el, amelyek kis módosítással, de az általánosított koncepciókat megtartva ültethetők át más platformokra is (pl. Java [98]).

Bármikor, amikor valamilyen újdonságot alkottam, mindig maximálisan szem előtt tartottam azt, hogy a tudományos szempontból hasznos eredmények az ipari szférában is alkalmazhatóak legyenek [10].

Ebből adódóan minden esetben fontosnak tartottam, hogy a kutatásom témakörét pontosan elhelyezzem a „konkurens” kutatások között, valamint rámutassak arra, hogy miért fontos, mitől több az, amit elértem. Szintén az előző bekezdésben leírt gondolatok kapcsán megjegyzem, hogy minden egyes elméletileg fontos eredményből egy gyakorlatban is használható terméket

állítottam elő.

Ezeket a dolgozatom megvédése után nyílt forráskódúvá [89], bárki számára elérhetővé kívánom tenni.

A dolgozat szerkezete az alábbi:

Az I. bevezető jellegű részben definiálom, hogy mit jelent a dolgozatban sokszor említett minőség és biztonság fogalma, ezek után felvázolom, csoportosítom azokat a módszereket, amelyek a programok minőségi és biztonsági jellemzőit javítják. Röviden kitérek a .NET keretrendszer alapszolgáltatásaira, alapinfrastruktúrájára, majd pedig megmutatom, hogy melyek azok a legfontosabb programminőséget növelő, valamint biztonsági szolgáltatások, amelyekkel a .NET keretrendszer alapkivitelben is rendelkezik.

A .NET legfontosabb nyelve a szabványos C# [84] nyelv, amely az objektum-orientált nyelvi elemek mellett deklaratív illetve funkcionális elemekkel is rendelkezik. Azonban a legtöbb népszerű programozási nyelvhez hasonlóan nem rendelkezik olyan szolgáltatásokkal, mint pl. a szofisztikált hozzáférés-vezérlés illetve a szerződés alapú tervezés [54]. Mivel az ebben a témakörben végzett kutatásaim részletes tárgyalása nem célja ennek a dolgozatnak, ezért csak röviden szólok az itt elért eredményeimről [7].

Amikor egy komplex alkalmazást készítünk, akkor olyan funkcionális (feladat által specifikált) illetve nem funkcionális (biztonsági, karbantarthatósági, stb.) követelményeket kell kielégíteni a rendszernek, amelyeket a monolitikus alkalmazásokkal, elavult fejlesztési módszerekkel már nem, vagy csak nagyon nehezen lehet teljesíteni. Kutatásomat abban a szoftver-architekturális [42], fejlesztési és módszertani környezetben végeztem, ahol ezek a problémák már nem kerülnek elő modernségükből adódóan. Ez a környezet a modern elosztott, szolgáltatás orientált, többrétegű alkalmazások kategóriája, amelyeket valamilyen iteratív, agilis [18] módszertan segítségével készítünk el.

Egy szoftverfejlesztési projekt során a specifikációs illetve a fejlesztési+tesztelési fázis során foglalkozunk leginkább szakmai, technológiai kérésekkel. Mivel ezek a fázisok egymásra épülnek, egymás után következnek, ezért a dolgozatomban először a specifikáció elkészítését segítő eredményeimet tárgyalom, majd pedig a fejlesztés+tesztelés támogatására térek át.

A II. részben egy rövid előkészítés után már saját eredményeket mutatok be az elosztott alkalmazások témakörében. A platform a korábban említett többrétegű, szolgáltatás orientált alkalmazásarchitektúrák. Ismertetem, hogy melyek azok a biztonsági szolgáltatások, amelyeket a legtöbb keretrendszer, így a .NET is támogat. Nyilvánvaló, hogy a keretrendszer segítségével készített és futtatott felügyelt programok egy operációs rendszeren, egy alkalmazás-szerveren,

egy tűzfal mögött futnak. Mindezen rétegek, hozzáadnak valamit ahhoz, hogy biztonságos környezetben működhessenek a programok.

Több kutató számára nyilvánvalóvá vált már, hogy ezek a szolgáltatások nem elégségesek. Ismertetem az általuk felvázolt koncepciók alapelveit, majd pedig rámutatok arra, hogy ezek a módszerek nem egy integrált rendszer létrehozását célozzák meg, hanem mint szigetrendszer üzemelnek. Ezek után egy komplex elosztott architektúrában azonosítom, illetve továbbfejlesztem azokat az építőelemeket, amelyek a biztonságosabb elosztott alkalmazások elkészítését szolgálják. Egy olyan formális módszert dolgoztam ki, amely segít ezeknek a biztonsági szolgáltatásoknak az integrációjában. Egy mondatban megfogalmazva: az üzleti szolgáltatásokhoz kötöm az alatta futó munkafolyamatokat; a felhasználói szintű, általam szabályalapú feltételekkel kibővített jogosultságkezelést; ezen felül a futás idejű hozzáférés-vezérlés egy elosztott alkalmazásokra való kibővítését is megalkottam. Először tisztán formális eszközök segítségével fogom definiálni a biztonsági szolgáltatások működését, hogy ezzel biztosítsam a megvalósíthatóság platform függetlenségét. A formalizmus használhatóságát ipari környezetből vett esettanulmányokon is tesztelem. Ez a formalizmus felfogható egy olyan megszorítási rendszerként, amely azokat a kritériumokat rögzíti, amelyek teljesülése esetén biztonságosabb, egyben magasabb minőségű programról beszélhetünk.

Az elkészült formalizmus gyakorlati felhasználására két utat látok:

1. A formalizmust egy átírási eljárás segítségével forrásnyelvi (esetünkben C#) attribútumokra írom át, majd ezeket futás időben ellenőrzöm.
2. A program futása során egy olyan részletes napló keletkezik, amelynek egy utófeldolgozó eljárás segítségével történő elemzése során a program futásának helyességét vizsgálni lehetséges.

Jelen dolgozatban az 1. pontot fejtettem ki, a 2. pontot, mint további kutatási irányt azonosítom.

Szintén a II. részben vázolom az említett formális keretrendszer egy lehetséges .NET alapú implementációját, amelyet a formalizmus validálásának céljából alkalmazom az előbb említett ipari példákra.

A programok minőségének javításához szükségünk van hatékony hibadetektáló, tesztelő és tesztgeneráló eszközökre. Ebből kifolyólag egy .NET környezetre specializálódott naplózó eljárást hoztam létre, amelyet a III. részben vázolok fel. Olyan nagy részletességű naplózási módszert ismertetek [2][9], amely:

1. Az I. részben felvázolt dinamikus programszeletelés [13] bemeneteként használható.

2. Futó programok hibakeresésének segítésére is alkalmazható.

Bemutatom azt az utat, amelyet végigjárva a végső megoldásig eljutottam. A .NET Profiler alapú megoldást egy két lépcsős iteráció során hoztam létre. Első lépésként a metódusok belépési és kilépési pontja valamint a szekvencia pont határok naplózására képes megoldást készítettem. Ez után ismertetem azt, hogy a meglévő koncepciókra alapozva, hogyan hoztam létre egy olyan megoldást, amely képes a tetszőleges változó olvasási és definiálási műveletet naplózni. A módszer részletességével egyedülállónak tekinthető a .NET platformon, ugyanis nem létezik más ismert naplózó megoldás a .NET platformra, amely képes lenne erre a szemcsézettségre.

Az általam kifejlesztett naplózási módszer abban is különbözik a többi megközelítéstől [33], hogy nem igényli sem manuálisan, sem automatikus eszköz segítségével a program eredeti forráskódjának módosítását ahhoz, hogy részletes naplót tudjon generálni futó programok esetében.

Végezetül olyan további kutatási lehetőségeket vázolok fel, amelyek a II. illetve a III. részben ismertetett módszereket terjesztik ki, integrálják.

I. Programok minősége és biztonsága valamint a .NET keretrendszer alapjai

1 A programok minőségéről és biztonságáról

Ebben a fejezetben definiálni fogom a minőség és a biztonság fogalmát, majd mindkettőnek meghatározom azt a részhalmazát, amelyet ebben a dolgozatban tárgyalni fogok.

1.1 A minőségről

Ahhoz, hogy a programok minőségéről, sőt csak magában a minőségről beszélhessünk, be kell vezetni a *minőség fogalmát*. Az ISO szabvány ezt a fogalmat a következőképpen definiálja: *"A minőség nem más, mint a szolgáltatás, illetve termék azon tulajdonsága, illetve jellegzetessége, hogy milyen mértékben felel meg a megrendelő, a felhasználó deklarált vagy feltételezett elvárásainak. " (ISO 9000:2005)* [83]. Ebből a definícióból ugyan kitűnik a minőségbiztosítási és projekt szemlélet, azonban ez nem zavaró tényező, hiszen az egyik célom az, hogy a program minőségét az alapján mérjem, hogy az mennyire felel meg az ügyfél által is elfogadott specifikációnak. A probléma a fenti definícióval az, hogy csak egy szemszögből vizsgálja a minőséget, mégpedig a felhasználó szemszögéből. A minőség definíciója függ a minőséget értékelő személy szerepkörétől, nézőpontjától illetve az értékrendjétől is. Általában 3 szereplő véleménye számít ebben a kérdésben: a felhasználóé, a fejlesztői és a menedzseré.

A *felhasználó* a szoftver használhatóságát, funkcionalitását, megbízhatóságát, hatékonyságát, hordozhatóságát, hibamentességét tartja szem előtt, és nem érdekli az, hogy milyen technológiát felhasználva és hogyan készült el. Ezzel szemben a *fejlesztő*t a karbantarthatóság, a tesztelhetőség, a kódminőség, a továbbfejleszthetőség, a kiterjeszthetőség érdekli. A *menedzser* abban érdekelt, hogy a szoftver minél előbb és költséghatékonyabban elkészüljön; valamint azt vizsgálja, hogy az ügyfél által elfogadott követelményeknek megfelel-e a szoftver.

Az egyik irány, amit ebben a dolgozatban követek az olyan fejlesztési módszerek és segédeszközök kidolgozása, amelyek hatékonyan képesek segíteni a fejlesztőt a tesztelés elvégzésében valamint a hibák felderítésében. Ebből kifolyólag hamarabb, kevesebb hibát tartalmazó programtermék állhat elő, amely egyúttal a felhasználónak és a menedzsernek is kedvez.

Számos módszer alakult ki, amelyek a programok minőségi mutatóinak javítását célozzák meg.

A **program szintézis** [36][68] során az elkészített absztrakt specifikációból kiindulva, ellenőrzött finomítási lépések segítségével állítjuk elő a programot.

A program helyességét, minőségét az biztosítja, hogy a finomítás lépései ellenőrzötten folynak. A módszer hátránya, hogy csak rövidebb programok

esetében alkalmazható hatékonyan. Előnye, hogy segítségével biztosítható, hogy a futó program megfelel a specifikációnak. Természetesen a specifikációban vétett hibák ellen ez a megoldás sem nyújt védelmet.

A **helyességbizonyítás** [68] éppen fordított irányból közelíti meg a helyes program előállításának folyamatát, mint a program szintézis. Előfeltételünk a specifikáció és az elkészült program. Az elkészült program egy absztrakt modelljén látjuk be, hogy megfelel a specifikációnak. Legtöbbször magas szaktudást igényel a helyességbizonyítás végrehajtása, illetve vannak megkötések azokra a programokra, amelyeken a módszer alkalmazható.

A módszer segítségével korrekt eredmény hozható létre. Hátránya, hogy általában csak egyszerűbb programokra alkalmazható.

A **Hibakeresés (Debugging)** leginkább a még fejlesztés alatt álló programon végzett nyomkövetési feladat, de már léteznek C++ metaprogramokra kidolgozott módszerek is [60]. A Debugging már a fejlesztési fázis fontos szereplője, amelyet a gyakorlatban a legtöbb program elkészítése során használunk. Itt a fejlesztő azt vizsgálja, hogy a program milyen lépéseket hajt végre, és közben melyik változók milyen értékeket kapnak.

Ez a módszer egy teljesen manuális feladatot takar, ahol csak és kizárólag a hibák javítása a cél. Tetszőleges programon elvégezhető a nyomkövetés a megfelelő Debugger eszköz felhasználásával.

A **futás idejű naplózás** nagyjából olyan viszonyban áll a Debugginggal, mint a programszintézis a helyességbizonyítással. Itt az elkészült programban elhelyezett ún. szondák által szolgáltatott futás idejű üzenetek [33], értékek adják a vizsgálat tárgyát. Megvizsgálhatjuk, hogy milyen függvények kerültek meghívásra, milyen változók milyen értéket vettek fel, milyen hibák keletkeztek, valamint milyen egyéb diagnosztikai üzenetek jöttek létre.

A lefutás után a napló manuális vagy automatizált vizsgálata szükséges.

A futás idejű napló elkészítése egy bizonyos szintig (pl. függvényhívás vagy kivételek jelzése) egyszerű feladat, azonban ha már utasítás vagy változóhasználat szintjén kívánjuk a naplót elkészíteni, akkor komplex metodológiákat kell alkalmaznunk. Ebből adódóan kevés ilyen részletességű napló létrehozását támogató rendszer létezik. A .NET keretrendszerre nincs ismert megoldás az ebben a dolgozatban felvázolt megoldáson kívül.

A **programszeletelés** [75][49] célja, hogy feltárja a program alkotóelemei közötti összefüggéseket, amellyel segíti a program fejlesztésének, megértésének folyamatát. Pontosabban a változók értékei és programutasítások lefutása közötti összefüggéseket vizsgálja. A programszeletelő algoritmusok informálisan

megfogalmazva azt a program szeletet (utasítások halmazát, sorát) adják vissza, amely egy adott utasításra közvetlenül vagy közvetve hatást gyakorolnak. A módszer tetszőleges programon alkalmazható. Két fajtája létezik: a *statikus* és a *dinamikus* programszeletelés. Weiser [75] eredeti koncepcióját Ottenstein szerzőpáros egészítette ki a PDG (Program Dependence Graph) [58] fogalmával, amely vezérlés- és adatfolyam analízis egy nagy mérföldkövének tekinthető.

A **statikus programszeletelés**i [71][3] algoritmusok csak a program forráskódját elemzik, nem veszik figyelembe annak lefutását. Az algoritmus által meghatározott program szelet azon utasítások halmaza, amelyek direkt vagy indirekt módon befolyásolhatják a *V* változóhalmazban található változók értékét a *p* programhelyen. Ezért a C=*(V,p)* párt szeletelési kritériumnak nevezzük.

Mivel ez a megoldás nem veszi figyelembe a program egy konkrét bemenetét és lefutását, ezért egy bővebb, a futási paramétereket figyelmen kívül hagyó eredményhalmazt ad.

A **dinamikus programszeletelés** [13][20][21][76] ezzel szemben a program bemenő paramétereit és a vizsgált utasítás lefutásának egy előfordulását is figyelembe veszi, ezáltal egy szűkebb, egzaktabb halmazt ad a vizsgált lefutásban befolyást gyakorló programutasítások körére. Itt a szeletelési kritériumot a következő hármas adja meg: *C=(I, o, V)*, ahol *I* jelöli a program bemenetét, *o* egy utasítás konkrét előfordulását, lefutását, *V* pedig azt a változóhalmazt, amit az *o*-ban vizsgálunk.

A dinamikus programszeletelés összekapcsolódik a futás idejű naplózással, ugyanis a szeletelési kritériumon felül szükséges ismernünk azt is, hogy mely utasítások milyen sorrendben futottak le, illetve mely változók mikor milyen értékadásokban, milyen szerepben vettek részt.

A **tesztelés** [45] egy olyan manuális vagy automatizált folyamat, amelynek során nem a specifikáció alapján dolgozunk, hanem az ebből készített tesztelési terv alapján.

A különböző teszteket különböző szerepkörben lévő emberek végzik el. A legfontosabb tesztelési módszerek a következők:

1. Unit (egység) tesztek – a program egységeinek, a program moduljainak a tesztelése oly módon, hogy ellenőrzi, hogy meghatározott bemenetekre azt a kimenetet kapjuk-e, mint ami az elvárt. A módszer szerint automatikus teszteseteket készítünk, amelyek később tetszőlegesen futtathatók.
2. Felhasználói felület tesztelése – azt vizsgáljuk, hogy a program felhasználói felülete segítségével elvégezhetők-e a specifikációban meghatározott műveletek. Ezen felüli cél az, hogy hibaüzenetek nélkül

fusson a program.

3. Integrációs tesztelés – az az igény, hogy a rendszerünk komponensei a specifikációban lefektetett módon, megfelelően kommunikálnak-e egymással.
4. Rendszer integrációs tesztelés – feladata annak a vizsgálata, hogy a mi rendszerünkkel kommunikáló külső rendszerek az elvárt módon illeszthetők, illeszkednek-e.
5. Teljesítmény tesztje – ennek a tesztnek az a célja, hogy megvizsgálja, hogy a rendszer adta válaszidők megfelelőek-e a különböző műveletek esetében.
6. Skálázhatóság tesztelése – itt azt vizsgáljuk, hogy ha több felhasználó egyszerre használja a rendszert, akkor az képes-e több és több erőforrást (CPU, memória, hálózat, tár) kihasználni. Illetve ezek bővítése esetében nő-e a párhuzamos teljesítmény.

A tesztek futtatása sokszor időigényes, azonban tetszőleges program esetében kivitelezhető.

Az automatikus (egység) tesztelő eljárásoknál sokszor az a probléma merül fel, hogy egy módosítást nem tesztelnek hatékonyan. Azaz egy módosítás hatásait nem tesztelik le (mert nincs olyan teszteset, amely ezt tesztelné), vagy pedig olyan kimeneti paramétereket vizsgálnak, amelyekre nem hat a módosítás.

A **hatásvizsgálat** segítségével megtalálhatók a módosításokat közvetlenül vagy közvetve befolyásoló bemeneti paraméterek és szignifikáns kimeneti értékek. Ennek az információnak a birtokában olyan tesztesetek is generálhatók, amelyek hatékonyan tesztelik a módosításokat. A hatásvizsgálat definíciója csaknem megegyezik a program szeletelés fogalmával. A különbség annyi, hogy a hatásvizsgálat esetében a program eredeti és módosított verziója esetében detektált eltérő utasításokra végzünk hatásvizsgálatot, a programszeletelés pedig egy programverzió esetében értelmezett fogalom. A hatásvizsgálathoz kapcsolódó eredményeimet [6] C++ környezetben értem el.

A Model-based testing azaz **modell alapú tesztelés** egy feltörekvő új filozófia. A programot először, mint (UML [63]) modellt írjuk le, amely alapján elkészítjük az alkalmazást. A modell segítségével absztrakt teszteseteket hozunk létre, amelyek még nem futtathatók a kész alkalmazáson, ugyanis, a tesztesetek absztrakciós szintje a modellével egyenrangú. Ezért az absztrakt tesztesetek átírása szükséges arra a szintre, ahol a programot is elkészítettük [39].

1.2 A biztonságról

Amikor a programok biztonságáról beszélünk, akkor a minőséghez hasonló nagy területtel találkozunk. A biztonság fogalma sok szerteágazó területet foglal magába: titkosító módszereket, (vírus)védelmi szoftvereket, vállalati vagy rendszer szintű házirendeket, jogosultságkezelést, szoftver szintű védelmi mechanizmusokat, kockázatelemzést, kockázatmenedzsmentet, stb.

Bruce Schneider [91] a következőt mondja a biztonságról: „A biztonság nem egy termék, hanem egy eljárás." [90]. Ez a megállapítás teljesen konform az *ISO 17799* szabvány [82] megfogalmazásával is. A szabvány definiálja azokat a követelményeket, eljárásokat, amelyek egy szervezet IT rendszerének biztonságosabbá tételéhez szükségesek. Ezek röviden a következők:

1. **Biztonsági szabályzat:** Egy szabályzatban kell körvonalazni a biztonsági elvárásokat, amely a vezetés részére iránymutatást ad a biztonságos működéshez. Ez egy szöveges dokumentum, amely az alapvető célokat, elveket tartalmazza. Bármely dolgozó számára érthető nyelven kell megfogalmazni.
2. **Biztonsági szervezet:** Meg kell határozni azt, hogy a szervezetben mely egység, a biztonság mely területéért felel. Legtöbbször megkülönböztetjük az olyan szervezeti egységet, amely az IT eszközök, kliens számítógépek biztonságáért felel. Van olyan szervezeti egység, amely a szervereket kezeli. Fontos a dolgozók biztonsági oktatása, illetve a portaszolgálat is lényeges szerepet tölt be a ki és bevitt informatikai eszközök nyilvántartásában.
3. **Biztonsági javak kezelése:** Osztályozni, leltározni kell a biztonsági javakat, amelyek azonosítás után a megfelelő védelemmel láthatók el. Azaz nyilvántartásba kell venni a hardver és szoftver eszközöket, ezeket kategorizálni kell, felelősöket kell hozzájuk rendelni.
4. **Személyi biztonság:** A dolgozóknak megfelelő biztonságtechnikai oktatást kell biztosítani, hogy tudatosan figyeljenek a biztonságra. Ez alatt általános biztonsági oktatás, illetve konkrét szoftverek használatához szükséges biztonsági oktatás értendő.
5. **Fizikai és környezeti biztonság:** A fizikai eszközök, berendezések védelme. Legegyszerűbb egy betörést akkor kivitelezni vagy adatokat eltulajdonítani, ha fizikailag is hozzáfér a támadó az eszközökhöz. Cél az, hogy ennek kockázatát minimalizáljuk, illetve ne tegyük lehetővé az érzékeny adatokhoz való fizikai hozzáférést.
6. **Kommunikáció és műveleti menedzsment:** Ez a leginkább

szerteágazó rész:

- Az információ feldolgozó egységek helyes és biztonságos működésének biztosítása. Mindez a rendszer felügyeletével megbízott szakemberek feladata az, hogy ezt biztosítsa.
- A rendszerhibák minimalizálása, amely folyamatos monitorozással és megelőző magatartással érhető el leginkább.
- A szoftverek és az információ integritásának védelme, amelynek az előzőekben megfogalmazottakon kívül feltételezi még a magas minőségű szoftverek használatát.
- Információfeldolgozás és kommunikáció integritásának védelme, amely a hálózati és adatfeldolgozó eszközök karbantartását, felügyeletét szorgalmazza.
- Információ hosszú távú őrzését támogató eszközök bevezetése. Olyan mentési illetve archiválási rendszerekről beszélünk, amelyekről katasztrófa esetében visszaállíthatók az adatok.
- A sérülések elkerülése illetve az üzlet folyamatos aktivitásának biztosítása. Ez az elv a megelőző magatartást szorgalmazza, amely leginkább a rendszer folyamatos monitorozásával és a biztonsági frissítések telepítésével érhető el.
- A szervezetek között mozgó információ elvesztésének, módosításának, vagy illetéktelenek kezébe kerülésének megakadályozása. Ez a pont a hálózati eszközök sértetlenségére, a kommunikáció illetve a tárolt adatok megfelelő titkosítására hívja fel a figyelmet.

7. **Hozzáférési jogosultság ellenőrzése:** A hálózati és alkalmazás interfészek hozzáférhetőségének korlátozása és a hozzáférés szigorú ellenőrzése a cél. Ezt a feladatot szofisztikált tűzfalmegoldásokkal, biztonsági címtárral, illetve alkalmazás szintű jogosultságkezeléssel oldjuk meg.
8. **Rendszerfejlesztés és karbantartás:** Minden (tovább) fejlesztést a biztonság szem előtt tartásával kell végezni. Fontos, hogy ne hozzunk létre biztonsági lyukakat egy rendszer kifejlesztése, továbbfejlesztése során. A biztonságot, már a tervezés során figyelembe kell venni.
9. **Az üzleti folytonosság menedzsmentje:** Az üzleti aktivitás megszakadása esetére legyenek előkészített eljárások, illetve legyen biztosított az alap üzleti folyamatok zavartalansága kiesés vagy

katasztrófa esetén. A PreDeCo [90] elv a következő hármast mondja: megelőzés (prevention), felismerés (detection) illetve elhárítás (correction). A legfontosabb az incidensek megelőzése. Amennyiben egy nem kívánt esemény bekövetkezett, akkor a minél előbbi felismerésre, valamint az elhárításra kell helyezni a hangsúlyt.

10. **Megfelelőség:** A szervezetnek vizsgálni kell, hogy mennyire felel meg az *ISO 17799*-nek vagy más hasonló, elfogadott szabványoknak. A rendszerek ilyen irányú folyamatos auditálása szükséges. Nagyon fontos megemlíteni, hogy nem a túldokumentáltság a legfontosabb momentum, amely az ISO szabványok esetében tipikus tévhit, hanem a folyamatok kiépítése.

A fenti pontok a szervezet, individuális személyek, a rendszert felügyelő biztonsági szakemberek és rendszermérnökök, illetve a szoftverek fejlesztését végző szoftverfejlesztők szerepét hangsúlyozza.

Mivel ez a dolgozat a szoftverek fejlesztésének biztonságosabbá tételét is megcélozza, ezért már most, a bevezető fejezetekben felhívom az olvasó figyelmét a fenti „tíz parancsolat" 7. és 8. pontjára. Ezek azok, amelyek a szoftver fejlesztésében résztvevő architetektek (szoftver építészek), vezető fejlesztők illetve fejlesztők alapvető gondolkodásmódját kell, hogy meghatározzák. A dolgozatban vizsgálni fogom azt, illetve módszereket adok a javítására annak, hogy a szoftver bizonyos kódrészletei milyen más program kódrészek által, illetve milyen hívási biztonsági kontextusban (milyen felhasználók, milyen kód jogosultsággal, milyen hozzáférést-vezérlési illetve hálózati megszorításokkal) futtathatók. Ezeket a feltételeket szigorúan fejlesztői szemmel veszem górcső alá.

Olyan módszereket adok, amelyek közérthetőségük miatt könnyen adaptálhatók, ugyanakkor hatékonyan képesek növelni a biztonságot.

2 A .NET keretrendszer alapjai

A .NET Framework (keretrendszer) [99] egy nyílt, szabványosított [81][85] fejlesztői platform, amely 2001-es bejelentése, majd 2002-es megjelenése óta nagy fejlődésen ment keresztül. Sok aspektusában hasonlít a Java [98][57] platformhoz, azonban a Microsoft által készített autentikus .NET futtatókörnyezet csak Windowsra létezik, azaz platformfüggő. Természetesen nemcsak kliens Windows kiadásokon (XP, Vista, 7) futtatható, hanem a szerver verziókon (Windows Server 2003, Windows Server 2008, stb.) is. Ezen felül egy speciális .NET kiadás képes mobil eszközökön is üzemelni, az itt rendelkezésre álló szűkösebb erőforrásokra optimalizáltan. Mivel egy szabványosított platformról beszélünk, ezért a Novel támogatásával létrejött a Mono [102], amely a .NET keretrendszer kevesebb tudású, platform független implementációja. Másik fontos implementáció a Rotor, a .NET keretrendszer nyílt forráskódú verziója, amelyet a Microsoft kutatási célokra készített.

A .NET előnye kifejezetten abból származik, hogy nyelv-független. Legfontosabb nyelve a szabványos C# nyelv [80][84][52], ami a jelen dolgozatban is sokszor központi szerepet fog betölteni. A nyelvfüggetlenség lehetővé teszi többféle programozási nyelven készült alkalmazásmodul együttműködését, azaz az egyik nyelven megírt alkalmazásmodul (.NET terminológiával: assembly, magyarul szerelvény) hívhat egy másik nyelven készült alkalmazásmodult. Ezt oly módon teszi lehetővé a .NET, hogy nem gépi kódra fordítja a forrásnyelvi programokat (C#, VB.NET, Managed C++, stb.) – ahol szigorú, platformfüggő hívási konvencióknak kellene eleget tenni -, hanem egy köztes nyelvre fordít, amelyet IL-nek (Intermediate Language) nevezünk.

Az IL kód egy assembly-jellegű, de ennél magasabb szintű platform-független nyelv, amelyet verem alapú műveleteket végrehajtó utasítások alkotnak. Az itt található utasítások az alapvető matematikai és logikai műveleteken kívül platform szinten kezelik többek között a tömböket, a memóriaallokációt, a dobozolást (boxing), sőt még a virtuális metódushívást is.

Ahhoz, hogy az IL kódot futtatni lehessen egy újabb fordítási lépésre van szükség, amelyet JIT-fordításnak (Just In Time compilation) nevezzük. A JIT fordítás után már egy platformfüggő (CPU architektúra, bitszám, stb) kód keletkezik, amely a konkrét platformra való optimalizáltságából adódóan, hatékony futást tesz lehetővé.

A JIT-fordítást a CLR (Common Language Runtime – Közös Nyelvi Futtatókörnyezet) végzi futás időben, mégpedig úgy, hogy minden egyes típus első elérésekor végrehajtja a típus szintű fordítási lépéseket (statikus változók

inicializálása, statikus konstruktor fordítása) valamint minden nem statikus metódus esetében az első futtatáskor történik meg a JIT-fodítás. A fordítások eredménye nem őrződik meg a program következő futtatásáig, azonban lehetőség van teljes modulok lefordítására és perzisztens tárba helyezésére. A .NET keretrendszer teljes alapkönyvtára JIT-fordítások átesett állapotban is telepítésre kerül, ugyanis ez tartalmazza azokat az alapvető műveleteket, amelyeket bármely programnak késedelem nélkül, azonnal futtatnia kell.

A CLR feladata nemcsak a JIT-fordítás elvégzése, hanem a

1. memóriakezelés,
2. rendszer erőforrások kezelése,
3. ellenőrzött futási környezet biztosítása,
4. típusellenőrzés,
5. automatikus szemétgyűjtés (GC – Garbage Collection) levezénylése,
6. kivételkezelés,
7. együttműködés megteremtése a régebbi, COM [35] technológiát használó modulokkal,
8. szálkezelés.

Az első négy pontról részletesebben a 2.1. fejezetben szólok.

A .NET szemétgyűjtő algoritmusa az olyan generációs szemétgyűjtő algoritmusok közé tartozik, amely a mark&compact (megjelölés&tömörítés) elvén működik [51].

Másik fontos fogalom a CTS (Common Type System – Közös Típusrendszer), amely azt definiálja, hogy a .NET-nyelvek típusrendszerének milyen alapvető követelményeket kell teljesítenie (számtípusok bitszáma, ábrázolási módja, karakterek, string-ek ábrázolása). A fenti követelményeket a CLS (Common Language Specification – Közös Nyelvi Specifikáció) foglalja össze.

Minden .NET szerelvény egyértelműen azonosítható egy négy tagú összetett kulcs segítségével (modul neve, verziója, kultúrája, erős nevű kulcs (strong name key)), amelyet erős névnek nevezünk. Az erős név (strong name) felhasználásával biztosítható az, hogy futtatáskor az a modul kerüljön felhasználásra, amit fordítási időben meghivatkozott a hívó modul. Ezáltal elkerülhető a hírhedt *dll hell*, amely a régi (Windows XP előtti) verziók natív modulokat használó programjait érintette. Egy szerelvény erős neve fordítás után már nem változtatható meg, ezzel biztosítva a rendszer integritását.

A .NET rendelkezik a GAC (Global Assembly Cache – központi

szerelvénytár) fogalmával, amelybe erős névvel rendelkező modulok helyezhetők el. Az itt található modulok (az összes .NET alapkönyvtárbeli modul is) megoszthatók több alkalmazás között. A .NET modulbetöltője (*fusion loader*) először a GAC-ból próbálja meg betölteni a programmodulokat még abban az esetben is, ha az a program privát könyvtárában is elérhető.

A .NET tartalmaz egy szerteágazó alapkönyvtárat (BCL) [51][14], amely a fejlesztők számára alapvető szolgáltatásokat valósítja meg. Ezeket röviden a következő táblázatban foglalom össze:

Szolgáltatáskategória	Szolgáltatások, technológiák
Alapszolgáltatások	alaptípusok kezelése, gyűjtemények kezelése, alapvető IO támogatás
Adatbázis elérés	ADO.NET, Linq2SQL, Entity Framework
XML	XML DOM és navigátor kezelése, validációk (XSD), transzformációk (XSLT, XPath)
Vastag kliens alapú felhasználói felület	Windows Forms, Windows Presentation Foundation
Vékony kliens alapú felhasználói felület	ASP.NET web platform, ASP.NET mobil platform, SilverLight
Kommunikáció	Alapvető socket alapú kommunikáció, Webszolgáltatások, Remoting, Windows Communication Foundation
Munkafolyamatok kezelése	Workflow Foundation
Hálózati szolgáltatások	IP alapú kommunikáció, alapvető protokollok (Ftp, http, Dns, stb.)
Nagyvállalati szolgáltatások	Message Queue (üzenetsor), COM+ támogatása
Biztonsági szolgáltatások	kriptográfia, címtár elérés,
Platformhívás	Natív WinAPI függvények, COM

1. táblázat: A .NET keretrendszer szolgáltatásai

A .NET köré egy kifejezetten nagy és hatékony közösség épült fel, amelynek tagjai egymás segítése mellett nyílt forráskódú alkalmazásokat is szabadon elérhetővé tesznek [92].

2.1 A felügyelt kód biztonsági vonatkozásai

Mint már említettem, a .NET különböző nyelvi fordítói nem gépi kódot generálnak, amelyet a számítógépbe szerelt CPU képes lenne végrehajtani, hanem egy köztes nyelvre az IL-re transzformálják a magas szintű programozási

nyelvekben elkészült programokat. A JIT fordítás során keletkeznek majd az adott platformra optimalizált gépi kódú utasítások. Tekinthetjük ezt egy köztes rétegnek is a programunk és az operációs rendszer között, hiszen az operációs rendszer szolgáltatásai a .NET futtató környezetén keresztül érhetők csak el. Ez a réteg különböző vizsgálatokat, ellenőrzéseket, biztosítékokat szolgáltathat arra, hogy a programunk nagyobb biztonságban futhasson. Azt, hogy milyen problémákat old meg a .NET a kódbiztonság terén, a következő példákkal kívánom szemléltetni:

1. A natív (C, C++, stb.) [66] metódusok hívása során bármilyen típusú paramétereket átadhatunk a metódusnak, amely leginkább a pointerek használata esetén lehet veszélyes. Helytelen paraméterátadás esetében a hívott eljárás megpróbálja értelmezni a kapott értékeket, címeket majd pedig vagy szegmenshibát kapunk, vagy nem megfelelő eredményt, legrosszabb esetben pedig olyan memóriaterületek kerülnek felülírásra, amelyek érzékeny adatokat is tartalmazhatnak.
2. A buffer overrun (puffer túlcsordulás) azt jelenti, hogy egy tömb vagy szöveg méretét kisebbnek gondoljuk (gondolja az eljárás), mint amekkora, így a kilógó részek, bájtok más adatokat fontos memóriaterületeket írhatnak felül. Ezáltal, ha a veremtartalmat módosítjuk, akkor akár az ott elhelyezkedő metódus visszatérési címet is megváltoztathatjuk, aminek hatására ártalmas kód futtatására adódhat lehetőségünk. Ez az a támadási forma, amit a legtöbb esetben kártékony célokra is ki szoktak használni.
3. A natív programok készítése során gyakori hiba az, hogy a programozó a *malloc* vagy ezzel rokon metódus illetve a *new* operátor segítségével létrehozott objektumokat, majd a lefoglalt memóriát nem szabadítja fel. Memóriaszivárgás jön létre. Ez leginkább a hívási lánc alján szereplő, gyakrabban meghívásra kerülő metódusoknál veszélyes, ugyanis ekkor képes a memória leginkább szivárogni. Hosszan futó alkalmazások esetében, amelyek olyan kritikus rendszerek alapszolgáltatásait adják, amelyek nem állhatnak le, a memóriaszivárgás akár végzetes probléma is lehet.
4. A natív alkalmazások minden egyes megnyitott objektumra egy handle-t (leírót) hoznak létre, ennek megadásával, alacsony szintű függvényhívásokkal kezelhetjük ezeket az objektumokat (fájlok, ablakok, stb.), amely számos hibalehetőséget hordoz magában.

Most azt mutatom be, hogy a .NET futtatókörnyezet, keretrendszer hogyan oldja meg a fent vázolt problémákat:

1. Felügyelt kód alkalmazása esetén biztosak lehetünk abban, hogy egy

metódust csak olyan típusú objektumokkal hívhatunk meg, mint amilyet a metódus definíciója engedélyes. A .NET ellenőrzi, hogy a paraméter futás idejű típusa megfelel-e a paraméter statikus típusának. Ezt a műveletet természetesen tetszőleges értékadás, kasztolás [57] esetében is elvégzi, nemcsak a metódushívások során. A nem biztonságos mutatók, pointerek fogalma ismeretlen a .NET keretrendszer számára. Természetesen a .NET lehetőséget ad natív WinAPI függvények elérésére, valamint COM hívásra, amikor is kilépünk a keretrendszer felügyelete alól. Ehhez azonban plusz jogosultságra van szükség.

2. A felügyelt végrehajtás során minden tömb és szöveg hossza ellenőrzésre kerül. Nem címezhetünk egy tömb végén túl, illetve nem használhatunk olyan karakterindexet, amely a szöveg végén túl helyezkedik el. Amennyiben ez megtörténne a .NET CLR egy kivételt dob.
3. A .NET keretrendszer rendelkezik egy szemétgyűjtő (Garbaga Collector) mechanizmussal, amely időközönként automatikusan felszabadítja a program által már nem használt memóriaterületeket (amire nincs már hivatkozás). Nincs olyan lehetőség, amelynek segítségével a fejlesztő explicit módon felszabadíthat egy-egy objektumot.
4. Mivel a .NET az operációs rendszert használja az alacsony szintű objektumok (fájl, ablak, stb.) elérésére, ezért minden ide kapcsolódó típus mögött is egy-egy leíró, handle található. Mivel a leírók kezelése illetve az ezeken végezhető műveletek magas szintű objektumokba, függvényekbe vannak csomagolva, ezért a beépített hibakezelés és szakszerű implementáció elrejti az alacsony szintű problémákkal való küzdelemből adódó hibákat. A programozók hatékonyságát is növeli, hogy egy magas szintű keretrendszer osztályaiba csomagolva érhetik el az alacsony szintű szolgáltatásokat.

A fentiekből látszik, hogy a .NET keretrendszer futtatómotorja számos olyan szolgáltatást ad, amely magasabb minőségű, hatékonyabban fejleszthető, biztonságosabb programok előállítását teszi lehetővé. A .NET futtatókörnyezet által kezelt kódot *managed* (menedzselt) kódnak is nevezi a szakirodalom.

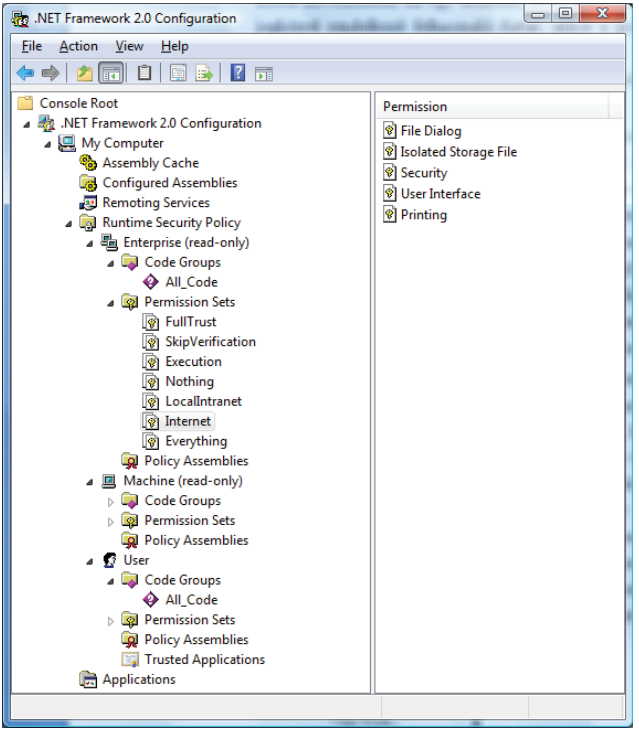
Még egyszer hangsúlyozom, hogy attól függetlenül, hogy az elkészült alacsony szintű kód egy felügyelt környezetben fut, még alkalmazás szinten számos biztonsági és minőségi probléma adódhat. Ezek egy részére kíván megoldást adni ez a dolgozat.

2.2 A .NET Kód Eredet Alapú biztonság

Az operációs rendszer nem tesz megkötéseket arra vonatkozóan, hogy milyen műveleteket hajthat végre egy lokálisan futó program vagy éppen egy olyan, amely egy másik számítógépről származik. Ez alól kivételt képeznek a böngészők (pl. Internet Explorer) amelyek korlátozásokat adnak a weboldalról letöltött programok esetében.

Mindig csak és kizárólag a futtató felhasználó jogkörei határozzák meg, hogy a felhasználó által indított program milyen műveleteket végezhet el (milyen fájlokat írhat és olvashat, milyen rendszerbeállításokat módosíthat, stb.). Ebből következően, ha egy ártalmas, távoli helyről indított programot egy magas jogkörrel rendelkező felhasználó futtat, akkor a program nem várt műveleteket végezhet. Látszik, hogy itt a problémát nem közvetlenül a figyelmetlen felhasználó okozza, aki nem megfelelő körültekintéssel indította el a kártékony programot, hanem maga a program. Ezt a szemléletmódot követi a .NET Kód Eredet Alapú (Evidence based security, Code Access Security (CAS)) [51] biztonsági megoldása, amely a program vagy programmodul származása szerint korlátozza a futó kódot. Pl. a helyi számítógépen futó alkalmazás tetszőleges műveletet elvégezhet, persze a 2.1. fejezetben említett megszorítások mellett. Sőt még arra is lehetőségünk van, hogy natív kódrészleteket hívhassunk meg, persze a megfelelő körültekintés mellett. A helyi hálózatról vagy az Internetről futó alkalmazások ennél kevesebb jogosultsággal rendelkeznek (nem érhetik el a fájlrendszert vagy csak korlátozott módon, nem nyithatnak ablakot, nem használhatják a teljes hálózatot, stb.)

Természetesen a fenti beállítások részletesen testre szabhatóak, azaz nemcsak az határozható meg, hogy az Internetről származó kódok mely műveleteket végezhetik el, hanem konkrét URL-re, konkrét assembly-re, kiadóra, elérési útvonalra (összefoglaló néven Code Group (*Kódcsoport*)) korlátozhatjuk ezeket a szabályokat. A szabályok megadhatóak *felhasználó*, *számítógép* és *szervezet* szinten is. A keretrendszer, amikor kiértékeli, hogy az adott programnak, kódnak milyen jogosultságai lehetnek, először minden szinten összegzi a jogosultságokat. Mindezt oly módon, hogy megvizsgálja, hogy a kódcsoportokra milyen jogosultsági halmazok adottak, majd ezeket összegzi. A különböző szinteken kiszámított összeghalmazoknak a metszetét veszi.



1. ábra: A .NET futás idejű biztonsági házirendje

Tehát a klasszikus, felhasználó szintű jogosultsági megkötések mellett, felett a program meghatározhatja, hogy milyen *minimális* jogosultságokra van szüksége a futáshoz, illetve azt is, hogy melyek azok az *opcionális* jogosultságok, amelyeknek a birtokában nagyobb funkcionalitással tud működni. Ezen felül lehetősége van *visszautasítani* azon jogosultságokat, amelyekre nincs szükség az alkalmazás futásához. A következő képlettel írhatjuk le a jogosultságok meghatározásának módját:

$$Jogosultságok = (Minimális \cup Opcionális) \cap Kalkulált - Visszautasított$$

Természetesen, ha a képlet által meghatározott jogosultságok halmaza nem tartalmazza a minimális jogosultságok halmazát, akkor a program nem fog futni, hanem biztonsági kivétellel leáll.

3 A modern elosztott alkalmazásokról

Az elosztott rendszerek folyamatosan bonyolultabbá és bonyolultabbá válnak. A rendszer architektúrája és azok az üzleti folyamatok, amelyeket implementálnak több hálózatot, számítógépet fognak át. Az sem biztos, hogy egy fajta programozási nyelvben készültek a rendszer moduljai. Azok a felhasználók, akik ilyen nagyméretű rendszereket igényelnek, leginkább olyan üzleti környezetben dolgoznak, amely folyamatosan és gyorsan változik. Az üzlet gyors változása és az új igények megjelenése természetesen a konkurenciával való versenyképesség fenntartása miatt elengedhetetlen. Az IT rendszereknek ezeket a változásokat rugalmasan, minél rövidebb idő alatt követniük kell. Ezek a változások nemcsak az üzleti logikát implementáló programkód, hanem a különböző integrált rendszerek egymáshoz kapcsolódó interfészeinek módosítását, illetve új interfészek implementálását is vetítik előre.

Ha programfejlesztési módszertanok szempontjából vizsgálnánk a kérdést, akkor azt mondhatnánk, hogy a pár éve még általánosan alkalmazott vízesés modell már nem elegendő, a spirális modell evolúciójának tekinthető agilis módszertanok (XP, Scrum, MSF4, stb.) [88] alkalmazása az üdvözítő.

Végül egy áttekintést adok arról, hogy milyen programozási paradigmákat használhatunk az alkalmazásaink kifejlesztésénél. Ezek közül melyiket hol használjuk, hogyan kapcsolhatók össze.

3.1 Szolgáltatás Orientált Architektúra

Néhány éve még komoly erőbefektetésre, manuális munkára volt szükség ahhoz, hogy a nem szabványos protokollon összedrótozott rendszerkomponenseket módosítsuk, teszteljük, elvégezzük a rendszer integrációját, lefuttassuk a rendszerintegrációs teszteket, majd pedig termelésbe állítsuk az új rendszert.

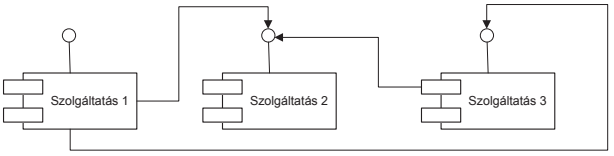
Természetesen feltehető a kérdés, hogy most mitől is lenne ez másképp. Mindenre választ ad az a módszertan és architekturális ajánláscsomag, amelyet röviden SOA-nak (Service Oriented Architecture – Szolgáltatás Orientált Architektúra) [19][106] nevezünk. Ha történeti szemszögből tekintünk a SOA-ra, akkor azt mondhatjuk, hogy a SOA a moduláris programfejlesztési módszertanok egy állomása, a moduláris programfejlesztés elveit viszi tovább. Ezek az elvek a következők:

1. Újrafelhasználhatóság – reusability,
2. Komponensekre bontás – componentization,

3. Összekomponálhatóság – composability,
4. Önállóság – autonomy,
5. Interopabilitás – interopability.

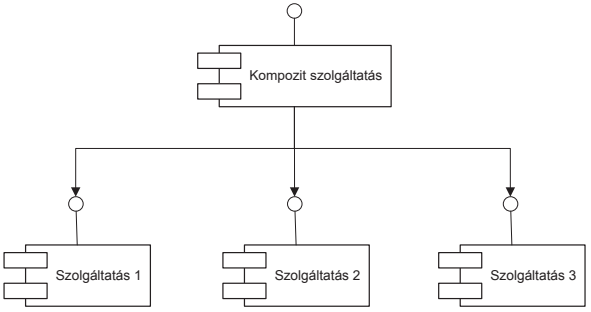
Az objektum orientált alkalmazások alapvető építőköve az osztály, a SOA alkalmazásoké ezzel szemben egy ennél nagyobb komponens, a szolgáltatás (service). (A komponensek publikus felülete mögött található implementációk alapvető építőkövei, mivel ezek maguk objektum orientált alkalmazások, szintén az osztály.) A szolgáltatások sokszor viszonylag nagy funkcionalitást implementálnak és egymástól a lehető leginkább függetleneknek kell maradniuk. Ez azt jelenti, hogy azt a logikát, amit egy adott szolgáltatásnak kell megvalósítania, lehetőleg minél inkább ez a szolgáltatás vezérelje, és ehhez ne használjuk fel további szolgáltatásokat. Amennyiben olyan komplex logikára van szükség, ahol elengedhetetlen több szolgáltatás által végzett munka, ott összetett kompozit szolgáltatásokat hozunk létre. Tehát a szolgáltatásoknak önállónak és összekomponálhatónak kell lenniük.

A fent említett negatív esetet, amikor egymás funkcionalitását keresztbehasználják a szolgáltatások, a következő ábra mutatja:



2. ábra: Helytelen szolgáltatáskompozíció

A következő ábra a kompozit szolgáltatásképzést mutatja:



3. ábra: Kompozit szolgáltatások

Mivel a második esetben nincsenek kereszthivatkozások, ezért ilyenkor könnyebb az implementáció, illetve egyszerűbb a karbantartás, a tesztelés is [40].

A szolgáltatások publikus felületét, interfészét szerződések formájában definiáljuk. Ezek a szerződések meghatározzák, hogy milyen szolgáltatás metódusok milyen paraméterezéssel használhatók, milyen visszatérési értékkel térnek vissza a hívás végén.

A szerződések definíciójával és az általuk definiált szolgáltatások mögöttes implementációjával szemben is elvárás a lazán csatoltság [42] elvének betartása. Egy komponens lazán csatoltsága azt jelenti, hogy a komponens környezetétől a lehető leginkább függetlenítjük a komponens implementációját. Ebben az esetben sokkal egyszerűbb az implementáció továbbfejlesztése, esetleg cseréje. Amennyiben lazán csatolt szolgáltatásokat használunk, akkor a szerződéstől teljesen függetlenné tudjuk tenni a szolgáltatás konkrét implementációját, ami az újrafelhasználhatóságot is növeli.

A modern elosztott rendszereknél elvárás az, hogy bármilyen programozási nyelvben készült komponens el tudja érni a szolgáltatásait, illetve bármilyen programozási nyelvben készült komponenssel ki tudjuk egészíteni a rendszer szolgáltatásait. Mindössze arra van szükség, hogy a komponensek interopabilisek legyenek, azaz valamilyen szabványos kommunikációs protokollt alkalmazzunk. Ez a protokoll leginkább a HTTP fölötti SOAP [79], azaz a web szolgáltatás (web service) szokott lenni.

3.2 A többrétegű architektúra üzleti alkalmazásokra történő adoptálása

Amikor egy kezdő programozó komplex rendszer fejlesztésébe kezd, ahol pl. adatbázisban található adatokon kell műveleteket végezni, valamint az adatokat felhasználói felületen megjeleníteni, akkor legtöbbször monolitikus, nem megfelelően tagolt alkalmazást készít.

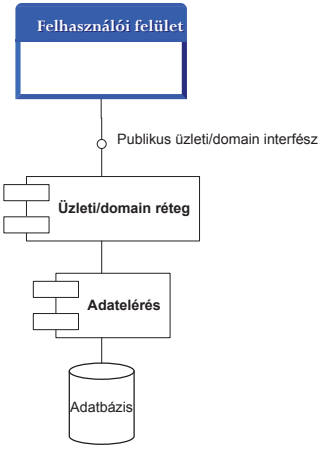
Ez alatt azt értem, hogy a fejlesztőeszköz, fejlesztőkörnyezet beépített szolgáltatásait felhasználva úgymond „összeklikkeli” az alkalmazást. Ebből kifolyólag minden feladat egy modulba fog kerülni, a felhasználó felülethez szorosan hozzá lesz kötve az alkalmazás logikája, valamint az adatbázis elérése.

Alapvető tervezési és fejlesztési elv, hogy az alkalmazásokat rétegekre bontjuk. Minden réteg egy specifikus feladatkört lát el. A felsőbb rétegek magasabb szinten értelmezett, absztraktabb funkcionalitással bírnak, az alacsonyabb szinten található rétegek pedig egyre alacsonyabb szintű egyszerűbb

műveleteket hajtanak végre. Egy adott réteg mindig az alatta levőben található elemi műveletekből építkezve komplex műveleteket komponál, illetve absztraktabb, egy felhasználó közelibb szinten értelmezi az adatokat, logikát, folyamatokat.

Amennyiben monolitikus, egyrétegű alkalmazást építünk, akkor az egyedüli modul, az egyetlen réteg túlzottan komplexszé válik. Többrétegű alkalmazások esetében minden réteg egyszerű, dedikált, jól meghatározott feladatkört lát el. Nyilvánvaló, hogy egy ilyen többrétegű rendszernek a kifejlesztése több időt vesz igénybe, azonban növekszik a rendszer átláthatósága, karbantarthatósága, automatikus eszközökkel való tesztelhetősége. A több rétegű rendszerek az objektumorientált programozásnál megismert kompozíció elvét absztrahálja.

A következő ábra magas szinten mutatja be a háromrétegű üzleti alkalmazás architektúrát:



4. ábra: Háromrétegű alaparchitektúra

Az adatokat legtöbbször *relációs adatbázisban* tároljuk, amely a modern megközelítés szerint egyre kevesebb, szinte semmi logikát nem tartalmaz. Az adatbázisok alkalmazásszintű feladata pusztán az adatok biztonságos tárolása, azok módosítása, valamint hatékony visszakeresésüknek támogatása.

Az *adatelérési réteg* feladata az adatok írása, módosítása, törlése, olvasása az alatta levő adatbázisból. Ez a réteg semmilyen logikát nem tartalmaz, általában olyan egyszerű műveletek absztrakt megvalósításával bír, mint pl. adatok táblába való beszúrása, egy adott azonosítóval rendelkező sor adatainak módosítása,

törlése, vagy éppen megadott azonosítóval rendelkező sor lekérdezése, illetve megadott feltételre illeszkedő sorok lekérdezése, stb.

Az *üzleti vagy domain réteg* ezen egyszerű műveleteket is felhasználva már olyan magasabb rendű műveleteket ad, amely a program megrendelője által elvárt funkciókat valósítja meg. Ilyen pl. egy számla rögzítése, felhasználói adatok módosítása, negyedéves kimutatás készítése a számlák alapján, stb. Az üzleti réteg nemcsak magának a műveleteknek a végrehajtásáért felelős. Vannak olyan kiegészítő szolgáltatások, amelyek az egész üzleti szolgáltatás réteget átszövik. Ezek legtöbbször a következők:

1. A műveletet végrehajtani kívánó jogosultságainak ellenőrzése, hogy van-e lehetősége végrehajtani a műveletet,
2. Diagnosztikai- és üzleti naplóbejegyzések létrehozása a rendszer működéséről.
3. A tranzakcionált adatkezelés [72] biztosítása

A *felhasználói felület* vagy *prezentációs réteg* felelős a felhasználóval történő ízléses, könnyen kezelhető felületen való kommunikációért. A felhasználói felületet úgy kell kialakítani, hogy annak segítségével a felhasználók a munkájukat a lehető leghatékonyabban, legkevesebb plusz „körrel", és lényegre törően el tudják végezni.

Az üzleti réteg fölé egy *publikus interfész*t helyeztem a fenti ábrán. Ez az az interfész, amelyen keresztül az üzleti logika szolgáltatásai elérhetők, amelyet szokás még *szerződés*nek, *üzleti homlokzat*nak is nevezni. Pontosan ez az az interfész, amelyről az előző fejezetben már beszéltem a SOA kontextusában.

Az előbbi ismertető alapján sok olyan kérdés merül fel, amelyek megválaszolásra szorulnak. Ezek a következők:

1. Milyen komponensekre érdemes felosztani a rendszer architektúra elemeit és miért?
2. Milyen interfészt definiáljunk a felhasználói felület és az üzleti logika között?
3. Milyen interfészt definiáljunk az üzleti logika és az adatbázis elérő réteg között?
4. Milyen kapcsolat van az adatbázis elérés és az adatbázis között?
5. Ha elosztott rendszerről beszélünk, akkor milyen módon történik a kommunikáció gép- illetve processzhatárok között?
6. Mi a legkevésbé fájdalmas módja az üzleti logika extra szolgáltatásainak, azaz a jogosultságkezelésnek, naplózásnak, illetve tranzakció-kezelés megoldásának?

A következő néhány bekezdésben bár nem tételesen, de a fenti kérdésekre fogok választ adni.

A többrétegű alkalmazásokkal szemben elvárás, hogy minden réteg csak a közvetlenül alatta levő réteget hívhassa (bizonyos kivételes esetekben akár több szinttel alatta levő rétegbe is történhetnek hívások). Ahhoz hogy ezeket az elveket kikényszerítsük, célszerű az adatelérést, az üzleti logikát illetve a felhasználói felületet is külön komponensbe elhelyezni. A felhasználói felület rendelkezzen (távoli) hivatkozással az üzleti logikára, az üzleti logika pedig az adatelérésre, ezáltal a felhasználói felület az üzleti logikát, az üzleti logika pedig az adatelérést hívhatja. Az üzleti logika és az adatelérés felfelé nem hívhat, ugyanis a komponens szintű kereszthivatkozások rendszerszinten tiltottak a modern platformokon.

Beszéltünk arról, hogy milyen programlogikát hova érdemes elhelyezni, de még nem esett szó arról, hogy milyen módon végezhető az adattovábbítás a rétegek között. Az *üzleti illetve domain entitások* azok a gyakorlatilag egyszerű osztályok, amelyek az adatok leírását végzik, illetve továbbíthatók a különböző rétegek között. Az adatbázisból származó adatentitásokat a különböző ORM [104] rendszerek alakítják át üzleti entitásokká és vissza. Ezek az entitások természetesen nem feltétlenül és közvetlenül adatbázis-bejegyzésekből származhatnak, hanem az üzleti logika által végzett műveletek eredményeként, paramétereiként is manifesztálódhatnak.

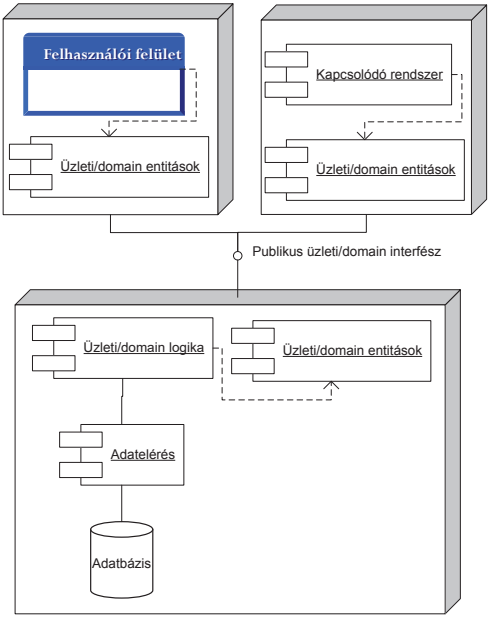
Mivel a felhasználói felületnek, illetve az esetleges kapcsolódó külső rendszereknek is ezekkel az entitásokkal kell dolgozniuk, ezért az *üzleti entitások* komponenst nemcsak az üzleti logikának (illetve gyakorlatban legtöbbször az adatelérésnek) kell ismernie, hanem a felhasználói felületnek illetve a külső rendszernek is. A gyakorlatban célszerűségi okok miatt ebben a modulban szokás elhelyezni a publikus interfészen publikált *szerződéseket* is, amelyek végső soron forrásnyelvű interfész típusok.

Végül, ha egy processzben fut mindegyik komponens kódja, akkor a komponensek közötti kommunikáció lokális metódushívássá egyszerűsödik. Amennyiben elosztott rendszerről beszélünk, akkor a felhasználói felületnek illetve a kapcsolódó rendszereknek távolról kell elérnie az üzleti logika szolgáltatásait. Ebben az esetben az üzleti homlokzaton publikált szerződéseket valamilyen szabványos felületen kell elérni, illetve az entitásokat valamilyen módon át kell juttatni a „drót" egyik oldaláról a másikra. Erre számos szabványos megoldás létezik. Ezek közül a legfontosabbak a webszolgáltatások (SOAP over HTTP) [79] illetve a WCF (Windows Communication Foundation) [108].

Az üzleti logika és az adatelérés az esetek legnagyobb részében egy

folyamatban fut. Az adatbázis eléréséhez a rendszerint külön folyamatban futó adatbázisszerver-alkalmazás többféle elérési módot biztosít, mint pl. TCP/IP, Named Pipe, RPC, stb.

A következő ábra grafikus formában is összefoglalja a fenti bekezdésekben megfogalmazott tudáscsomagot:



5. ábra: Elosztott rendszerek alaparchitektúrája

Nem szóltam még azokról a szolgáltatásokról (naplózás, jogosultságkezelés, tranzakcionális műveletvégzés, stb.), amelyet az üzleti logikának extraként kell elvégeznie az üzleti műveleteken kívül. Egy fontos tervezési elv, hogy az infrastruktúra-szolgáltatásokat az infrastruktúra-szolgáltatás felhasználójától szeparáltan, célszerűen külön komponensben implementáljuk, valamint minél jobban rejtsük el őket. Erre az elvre a későbbiekben még részletesebben kitérek.

3.3 A modern programtervezési, programfejlesztési módszertanokról

Az IT rendszerek illetve a megrendelői követelmények gyors változásának köszönhetően, a 3.1-ben illetve a 3.2-ben felsorolt architekturális igények, ajánláscsomagok kerültek előtérbe. Az ügyfél oldali versenyképesség fenntartása érdekében olyan rendszerfejlesztési és tervezési módszertanok alakultak ki, amelyek filozófiájukból adódóan képesek ezeket az igényeket kielégíteni.

A régi, alapvetően vízesés modellre [63] alapuló módszertanok háttérbe szorultak és a spirális modell leszármazottjai kerültek előtérbe. Azonban még a mai napig sok szoftverfejlesztő cég a külső igények vagy pedig belső szabályozások által gúzsba kötve a vízesés modellt kell, hogy kövesse.

A vízesés modell a következő fázisokat definiálja egy szoftverprojekt tekintetében:

1. Követelmények rögzítése,
2. Rendszertervezés,
3. Implementáció és a részegységek tesztelése,
4. Integráció és a rendszer tesztelése,
5. Bevezetés, oktatás, üzemeltetés, karbantartás.

További megkötések, hogy a fenti munkafázisok nem párhuzamosíthatók, hanem szigorúan egymás után következnek. Amennyiben menet közben változás történik a követelményekben, a folyamat elejére kell visszalépni. Ezáltal minél későbbi fázisban szembesülünk egy megváltozott követelménnyel, annál költségesebb a szoftver elkészítése.

A tervezés során sokszor több száz, rosszabb esetben több ezer oldal dokumentáció is keletkezhet. Ezekből a részletes tervekből nehézkes a munka, valamint mire elkészül a rendszer, már régen elavulttá válik. Az üzlet nem lesz versenyképes.

A szoftverfejlesztésnek, mint szakmának közel 30 éves története alatt számos szoftverfejlesztési módszertana alakult ki, majd tűnt el. Számos projekt vált sikertelenné, ezért a fő cél mindig a sikeresség rátájának növelése volt. Sajnos a különböző, nagyrészt sikertelen (Structured Systems Development, IDEF, SSADM, OMT, Objectory, Catalysis, OPEN, Unified Process) módszertanok láttán azt a következtetést vonták le, hogy ezek nem elég szabályozottak és precízek. A Unified Process leírása már több, mint 1000 oldalra rúg.

Valószínűleg ez azért történhetett meg, mert a szoftverfejlesztést a mérnöki tudományokhoz hasonlították, ezzel feltételezve a végtermék precíz

tervezhetőségét. Egy jó építészmérnök vagy gépészmérnök tud olyan épületeket illetve berendezéseket tervezni, amely a precíz tervek alapján gyártásba vihető, majd a megadott darabszámmal, minőségi kritériumok mellett legyártható, illetve felépíthető.

A szoftverfejlesztés ezzel szemben nem gyártás, a tervek nem adhatók oda gyártórobotnak képzelt fejlesztőknek. A szoftverfejlesztés minden alkalommal egy új terméket hív életre. Alapfeladat az, hogy a megrendelő és az üzlet által felvázolt problémára, mindig a legmegfelelőbb szoftver készüljön el.

Az előbbi, a gyakorlatban felmerült problémák megoldásaként a spirális modell leszármazottjait alkalmazhatjuk, amelyek legtöbbször az agilis módszertanok [88] (XP [18], Scrum [61], MSF4 for Agile [103] , stb.) közül kerülnek ki. Az agilis módszerek esetében kisebb (2-3-4 hetes) iterációkra bontjuk a fejlesztést, rendszeresen megmutatjuk az ügyfélnek a szoftvertermék aktuális állapotát, majd az újabb iterációba fogunk. A szoftvert akár már a fejlesztés során is el lehet kezdeni használni. Természetesen ahhoz, hogy egy fejlesztés alatt álló rendszert használatba lehessen venni, és gyorsan reagálni lehessen a változásokra, biztosítani kell a termék folyamatosan magas minőséget.

A következő alapelvek azok, amelyek a fent vázolt új gondolkodásmód alapján keltek életre:

1. **Legtöbb értéket képviselő termékek előállítása**. Olyan termékek előállítását célozzuk meg, amely a végső szoftvertermék szempontjából releváns és gazdaságos. Azaz lényegre törő rendszertervezést végzünk, és minimalizáljuk a dokumentációk méretét, körét.
2. **Test Driven Development** [18], azaz teszteléssel támogatott fejlesztés, amely azt jelenti, hogy minden egyes elkészült programegységhez egységteszteket (unit test) készítünk a fejlesztés minél előbbi szakaszában. Amennyiben valamilyen változást kell a rendszerben alkalmazni, az automatikusan futtatható egység tesztek megvédenek attól, hogy a már működő funkciókat elrontsuk. Hasonlóan fontos a felhasználói felület teszteseteinek rögzítése és többszöri elvégzése.
3. **Domain Driven Design** (DDD) [32], azaz a szakterület alapú tervezés során alapkövetelmény, hogy minden fejlesztő ismerje meg a megrendelő szakterület specifikus fogalmait, megoldandó problémáit, gondolkodásmódját, és ezáltal alakuljon ki egy közös nyelv a megrendelő és a fejlesztők között. A tervezés során az üzleti fogalmakra, lépésekre, funkciókra, folyamatokra kell koncentrálni. A

domain szintű ismereteket célszerű már az elejétől fogja entitások, interfészek formájában rögzíteni. Ez a domain modell, amely a fejlesztés során folyamatosan fejlődik, alakul. A forráskódnak ezt a szakterület alapú megközelítést kell mutatnia.

4. A **Refaktorálás** (refactoring) [34] [18][48] az a folyamat, amikor a forráskódot újastrukturáljuk, ésszerűsítjük. A refaktorálás egy jól bevált, mindennapos feladat, illetve folyamat kell, hogy legyen. Vigyázni kell természetesen arra, hogy a program működése a refaktorálás által még véletlenül se változzon meg. A TDD alapú megközelítés miatt bátran bele lehet vágni a refaktorálásba, ugyanis ha valamit elrontunk, akkor azt a tesztesetek nagy valószínűséggel jelzik.
5. **Páros programozás** (pair programming) [18] során egy számítógépnél két fejlesztő dolgozik. Az egyik fejlesztő az effektív kódírást végzi, a másik pedig tanácsokkal látja el őt. Időközönként cserélnek. Ennek a módszertannak az előnye abban rejlik, hogy magasabb minőségű kód állítható elő a több szem többet lát elv alapján. Ezen felül, mivel több programozó is behatóan megismeri ugyanazt a kódrészletet, ezért a későbbi továbbfejlesztést többen tudják hatékonyan elvégezni.
6. **Iteráció**k. Az iterációk [61] néhány (általában 2-3-4) hetes periódusokat, projektszakaszokat jelent. Az iterációk elején kitűzzük a lefejlesztendő funkcionalitásokat, szükséges szinten megtervezzük őket. Ezután következik az automatikus tesztesetekkel támogatott fejlesztés, majd az addig elkészült (lehetőleg stabilan működő) funkcionalitás bemutatásra kerül az ügyfélnek. Az ügyfél így nem csak a több hónapos, esetleg több éves projekt végén elkészült terméket tekintheti meg, hanem folyamatos rálátása van az aktuális állapotra. Minden iteráció elején figyelembe kell venni az előző iteráció során felmerült ügyféligényeket, megjegyzéseket.
7. **Folyamatos integráció** [18][42] (continuous intergration). A vízesés modellben a rendszerintegráció egy külön projektszakaszként tekintendő. Az agilis módszertanok ezzel a szemben folyamatos integráció elvét javasolják, amely azt jelenti, hogy a fejlesztés teljes időtartamában a rendszer minden komponensét egymással összecsiszolva is tesztelni kell. Ennek előnye az, hogy az integrációs problémák a projekt egy korai szakaszában előjönnek, illetve megoldásra kerülnek.

8. **Bizalom az ügyfél és a fejlesztő között**. Mivel ezek az agilis megközelítést sugalló módszertanok nem a papírgyártásra törekednek, ezért nagyon fontos, hogy a megrendelő és a fejlesztőcég maradéktalanul megbízzon egymásban. Mivel legtöbbször megelégszünk a funkcionális specifikációval, mint dokumentációval, ezért nem áll rendelkezésünkre annyi írott dokumentáció, amennyi az esetleges konfliktusok során a probléma megoldását segítené.
9. **Egyszerűségre való törekvés**. Nagyon fontos az, hogy egy adott probléma elemzése során mindig azt a megoldást válasszuk, amely a legegyszerűbb, és képes megoldani azt. Ez a „Do The Simplest Thing That Could Possibly Work" – DTSTTCPW elve [18]. Ennek az elvnek egy korábbi megjelenése a „Keep It Simple, Stupid." (Tartsd meg egyszerűnek, butának!) kijelentés, amelyet röviden KISS elvnek nevezünk. Ezzel párhuzamos fogalmak a rugalmasság, a könnyű bővíthetőség, a lazán csatoltság (louse coupling) is.
10. **Az infrastruktúra és a program logika szétválasztása**. Amikor magának az üzlet által felvázolt problémának a megoldását implementáljuk, akkor nyilvánvalóan külső adatforrásokkal (adatbázis szerver, webszolgáltatások, perifériák, stb.) történő kommunikációval is foglalkozni kell. Nagyon fontos alapelv az, hogy ezeket az infrastrukturális programrészeket maximális szinten elválasszuk a program logika kódjától. Tehát pl. nem akarunk adatbázis-kapcsolatokkal illetve tranzakció-kezeléssel alacsony szinten foglalkozni akkor, amikor más, súlyosabb problémák várnak megoldásra. Ebben segít a későbbiekben (3.4) megemlítésre kerülő AOP [46] is.
11. **Magasan képzett, nagy szaktudású fejlesztők**. Ahhoz, hogy a fenti elveket betarthassuk, minden esetben magasan képzett, lelkes, nagy szaktudású fejlesztőket kell alkalmazni. Olyanokkal, akik nemcsak kód szinten képesek gondolkodni, hanem az adott üzleti problémát is képesek magukévá tenni és a legmegfelelőbb módon megoldani azt.

Az előbbi néhány pontban csak azokat legfontosabb modern szoftverfejlesztési alapelveket említettem meg, amelyek a legkardinálisabb problémák megoldását célozzák meg az említett hiányosságok közül.

Természetesen a fenti elvek részletes kifejtése több száz oldalt is felölelne. Mivel ennek a dolgozatnak nem célja a szoftverfejlesztési módszertanok tárgyalása, ezért szorítkoztam erre a néhány oldalra.

3.4 A programozási paradigmákról

Ismertetem a legelterjedtebb programozási paradigmákat, valamint kiemelem, hogy melyik hol alkalmazható a .NET alapú fejlesztések során.

Nyilvánvaló, hogy egy-egy programozási nyelv [57] nem tisztán egyik vagy a másik paradigmát támogatja, hanem egyszerre többfajta elvet is alkalmazhatunk, amelyek között átjárhatóságot is biztosít a nyelv.

1. Imperatív programozás: A programozási nyelvek nagy hányada az imperatív, azaz utasításalapú paradigmára épül [36]. Lényege, hogy a programot alapvető elemi programokból a programkonstrukciók (szekvencia, elágazás, ciklus) segítségével nagyobb programokká szervezzük. Gyakorlatilag azt fogalmazzuk meg, hogy hogyan, milyen lépések sorozatával oldjuk meg az adott problémát.

2. Objektumorientált paradigma: A programot egymással együttműködő, saját felelősségi és szerepkörrel bíró, legtöbbször a való világot szimbolizáló osztályok, objektumok egymással együttműködő csoportjaként írja le. Az osztályok magukban foglalják az adatokat és az azokon végezhető (sokszor imperatív szemléletű) műveleteket is. Az objektumok és az őket meghatározó osztályok definiálása során olyan elveket [55] alkalmazunk, mint az információ beágyazása, az öröklődés, a polimorfizmus [23][65] vagy éppen a modularitás. A legelterjedtebb .NET-nyelvek (C#, VB.NET, stb.) alapvetően ebbe a kategóriába tartoznak.

3. Deklaratív programozás: Míg az imperatív és objektumorientált programozás a „hogyan?" kérdésre felel, ezzel szemben a deklaratív programozás [11] a „mit?" kérdésre keresi a választ. A deklaratív szemlélet formálisabb módszerekkel közelíti meg a kérdéseket. Tipikus példa az SQL adatlekérdező és definíciós nyelv, vagy éppen a XSLT nyelv, amelyet XML dokumentumok transzformációjára használunk. Fontos megemlíteni, hogy deklaratív elemeket fedezhetünk fel a .NET-nyelvek (pl. C#, Visual Basic) által használt attribútumokban [14] is. A deklaratív definíció mögött sokszor egy imperatív feldolgozó áll, amely működést ad a deklaratívan megfogalmazott információnak, tudásnak.

4. Generatív programozás: Az objektum orientált programozás kiegészítőjeként tekinthető paradigma [27][28] célja a programozási folyamat minél hatékonyabb automatizálása. A többféle generatív módszer közül a legfontosabb a metaprogramozás illetve a futás vagy előfeldolgozó lépésben végrehajtott automatizált kódgenerálás. Az aspektus-oritentált programozás a generatív programozás egy alfajának tekinthető, azonban jelentőségéhez mérten

külön pontban tárgyalom.

5. Aspektus-orientált paradigma (AOP): Az objektumorientált programfejlesztés során a legtöbb működési elemet modulok, objektumok, komponensek felelősségi körébe soroljuk. Azonban vannak olyan működési elemek (concerns), amelyeket nem tudunk konkrét építőelem felelősségi körébe sorolni. Ide tartozik például a naplózás (üzleti, de leginkább diagnosztikai), biztonsági szolgáltatások, automatikus tranzakció-kezelés, stb. Az aspektus-orientált programozás [46] azt célozza meg, hogy a modulok között átívelő ortogonális működési elemeket (cross-cutting concerns) a programok jól meghatározott helyén definiálhassa, majd pedig ráakassza azokra az objektumokra, műveletekre, ahol ezek szükségesek. Az AOP lehetővé teszi, hogy a programozó által meghatározott illesztési pontokhoz (pointcut) tetszőleges kiegészítő kódrészletet (aspect) szőjünk hozzá (weave). A legtöbb programozási nyelv nem ismeri az aspektus-orientált paradigmát, csak kiterjesztésként adható hozzá a nyelvhez.

6. Komponens alapú programozás: A komponens alapú programozás [67] az objektumorientált paradigma egy továbbfejlesztett változata. A paradigma alapötlete, hogy az egyszer már elkészített szoftverkomponenseket ne kelljen újra elkészíteni, hanem azok újrafelhasználhatóvá, kicserélhetővé váljanak. A funkciók jól definiált interfészen érhetők el. A paradigma segítségével csökken a rendszerek, modulok integrációjának ideje és költsége, biztonságosabb, gyorsabban alkalmazkodó, sokszor skálázhatóbb alkalmazások hozhatók létre.

7. Funkcionális programozás: A funkcionális paradigma [69][47][11][12][64] a deklaratív programozás egy letisztultabb változata. Egy imperatív program belső állapotinformációkat olvas és ír, a funkcionális programozás ezt a függvények használatával kerüli el. Ez a módszer sokkal kevesebb hibalehetőséget hordoz magában, és sokszor átláthatóbb algoritmusokat eredményez. A C# nyelv 3-as verziója [50] számos funkcionális elemet alkalmaz (lambda kifejezések, type inference, currying, lazy adatszerkezetek).

Szintén nagy jelentőséggel bír még a generikus programozás [15][38], valamint az előbbiektől nagyban eltérő szimbolikus programozás, illetve a logikai programozási nyelvek.

II. Elosztott alkalmazások biztonsága és a futás idejű hozzáférés-vezérlés

4 Elosztott alkalmazások biztonsági és minőségi kérdései

Ebben a fejezetben olyan biztonsági kérdéseket említek, amelyek akarva vagy akaratlanul megtalálhatók a legtöbb elosztott alkalmazásban illetve, ha megoldásra kerültek, akkor nem feltétlenül a legszakszerűbben. Ezek után megmutatom, hogy milyen kapcsolódó munkák születtek más szerzők tollából ebben a problémakörben.

A saját illetve a kapcsolódó munkákban található problémafelvetéseket figyelembe véve részleteiben kifejtem az elvárásokat egy elosztott rendszerekre alkalmazandó hozzáférés-vezérlő motorral kapcsolatban.

Az elvárások alapján egy formális modellt alkotok, amely a problémák absztrakt megoldását adja [8]. Ennek segítségével absztrakt szinten megfogalmazhatók az elosztott rendszerekre vonatkozó biztonsági megkötések.

4.1 A futás idejű hozzáférés-vezérlés és a szerződés alapú tervezés

Jelen dolgozatnak nem célja, hogy a nem elosztott alkalmazások esetében részleteiben tárgyalja a hozzáférés-vezérlés és a szerződés alapú tervezés lehetőségeit és az itt elért eredményeimet. Ebből kifolyólag csak igen röviden, probléma felvetés szintjén teszek róluk említést.

Kutatásom során foglalkoztam azzal is, hogy olyan módszereket adjak standard C# nyelvű nem elosztott programok minőségi mutatóinak javítására, amelyek már bizonyos környezetekben megoldottak, de a .NET keretrendszer illetve a C# nyelv nem ad lehetőséget használatukra:

1. Szofisztikált hozzáférés-vezérlés (Access Control)
2. Szerződés alapú tervezés (Design By Contract – DBC)

Az Eiffel programozási nyelv [54] és a hozzá tartozó futtatókörnyezet rendelkezik ezekkel a tulajdonságokkal.

Az egységbe zárás (encapsulation) [55] az objektum orientált programfejlesztési paradigma egyik legfontosabb elve. Amennyiben csökkentjük (és egyben racionalizáljuk) azokat az interfészeket, amelyeken keresztül a szoftver komponensek kommunikálhatnak egymással, akkor növeljük a szoftver minőségét és csökkentjük a fejlesztési költséget. A fordítás vagy futás idejű láthatóság vizsgálat vagy hozzáférés-vezérlés minden modern programozási nyelvnek és futtatókörnyezetnek része. Ezek a megoldások maghatározzák a szoftver komponensek felelősségi köreit, implementációs és biztonsági házirendeket

(policy) definiálnak. A legtöbb modern programozási nyelv [57], mint pl. a C++, a Java illetve a C# nem rendelkeznek szofisztikált hozzáférés vezérlési megoldásokkal, azaz csak a *public*, *private*, *protected*, *internal* illetve a *friend* módosítók részhalmazát illetve kombinációját használhatjuk. Az Eiffel ezzel szemben rendelkezik egy szofisztikált megoldással, amelyet selective exportnak (szelektív kiajánlásnak) nevezünk. Ez annyit jelent, hogy minden egyes (Eiffel terminológiával élve) feature (pl. függvény, változó, stb.) esetében meghatározható, hogy pontosan milyen típusok (osztályok) érhetik el az adott feature-t. Ez egy sokkal finomabb granularitással konfigurálható megoldás, mint pl. az, amikor a *public* kulcsszóval mindenhonnan hozzáférhetővé teszünk egy osztály valamely olyan tagját, amelyet csak egy kliens számára kívánunk elérhetővé tenni.

Hoare és Dijkstra [30][41] még az 1970-es, 1980-as években tette le az előfeltételeit a Design by Contract-nak, amelynek a lényege a következő: a specifikáció és a kifejlesztett program közötti egyezőséget nagyon nehéz biztosítani. Erre az egyik módszer az, hogy a programra, metódusokra elő- és utófeltételt adunk meg. Hoare a következő formalizmust adta a problémára:

$$\{P\}\ S\ \{Q\},$$

ahol

1. P logikai kifejezés (előfeltétel),
2. Q logikai kifejezés (utófeltétel),
3. S program, modul vagy metódus.

A program akkor felel meg a specifikációnak, ha P igaz és S terminál, akkor Q-nak igaznak kell lennie.

További feltételek adhatók még S-re:

4. Adott utasítások után megfelelő állapotban van a programunk,
5. Ciklus invariánsok teljesülése,
6. Induktív feltételek a ciklusokra.

Több módszertan is kifejlődött a fentiből, mint pl. [36].

A fenti, címszavakban bemutatott elmélet alapján *Bertrand Meyer* létrehozta a Design By Contact [53] fogalmát, és különböző kulcsszavak alkalmazásával felkészítette az Eiffel nyelvet arra, hogy a függvényekhez, osztályokhoz, stb. elő- és utófeltételeket valamint invariánsokat adhasson meg a programozó.

A legelterjedtebb programozási nyelvek, mint pl. C++, Java, C# nem bírnak az Eiffel-ben már rendelkezésre álló tulajdonságokkal.

Kutatásom során foglalkoztam olyan módszerek [4] [5] [7] kidolgozásával,

amelyek segítségével futás időben megvalósítható az Eiffel-szerű selective export funkcionalitás. Az eredmények természetükből, működésükből adódóan kiterjeszthetők úgy, hogy C# nyelvű programokhoz is adhassunk elő- és utófeltételeket valamint invariánsokat.

Az egyik megoldás erre a problémára az, hogy a .NET keretrendszer futás idejű híváselkapás [51] mechanizmusát kihasználva, az AOP [46] elvekkel integrálva hozzáadjuk a rendszerhez ezt a funkcionalitást. Ennek hátránya az, hogy nem mindig nyújt megfelelő teljesítmény, kiemelten rövid lefutású műveletek esetében, valamint kötelező a ContextBoundObject osztályból származtatni a kiterjeszteni kívánt osztályt. Ez nagy megkötés, ugyanis a C# nyelv nem támogatja a többszörös öröklődést.

A másik megoldás a C# 3 [11] nyelv extenstion method (kiterjesztő metódus) nyelvi elemének segítségével, és a fordító egy speciális működését kihasználva, az eredeti metódusok elrejtésével, majd azok kiterjesztő metódusokkal való meghívásával dolgozik. Ennek előnye az, hogy nagyobb teljesítményt ad az első megoldáshoz képest, valamint nem igényli azt, hogy a módosított osztályt egy dedikált ősből származtassuk. További pozitívumként említem meg, hogy ezek a kiterjesztő metódusok sablonos felépítésükből adódóan generatív eszközök [27] segítségével is generálhatók.

4.2 A klasszikus futás idejű hozzáférés-vezérlés kiterjesztésének lehetőségei

Ismert, hogy a szolgáltatások szerződéseken, interfészeken keresztül publikálják a publikus funkcionalitásukat. Mivel az interfészek teljesen nyilvánosak, ezért bármilyen hívó bármikor meghívhatja a mögöttes szolgáltatásokat. Elég nehéz feladat hozzáférés-vezérléssel felvértezni ezeket.

A gyakorlatban leggyakrabban a hívó felhasználó kilétét, csoporttagságát szoktuk ellenőrizni. A modern környezetekben ezt a feladatot megoldhatjuk deklaratív illetve imperatív módszerekkel.

A behívást végző kliens oldali komponensre vonatkozó megkötések kezelésére nem létezik ismert elméleti illetve gyakorlati megoldás.

Egy szolgáltatás által publikált üzleti metódusok azonban sokszor egy üzleti folyamat lépéseit vezérlik. Ebből kifolyólag azt is vizsgálni kell, hogy egy adott művelet egyáltalán értelmezett-e a futó üzleti folyamat aktuális szakaszában, állapotában. Erre a feladatra szintén léteznek gyakorlati megoldások, azonban közel sem olyan színvonalon, mint ahogy az elvárható lenne.

Célszerű még vizsgálni azt is, hogy milyen hálózati szegmensből történik a

szolgáltatás meghívása. Nem biztos, hogy minden metódust ugyanabból a hálózati szegmensből elérhetővé kell tenni.

4.3 Kapcsolódó munkák

Sok kutató foglalkozik elosztott alkalmazások biztonságával [22] [17] [25] [31] [74], ami jelzi a témakör fontosságát.

A kutatói munkát két [110] alapvető módszertani kategóriába sorolhatjuk. Az egyik az *analizáló* a másik pedig a *szintetizáló* módszer. Az analizáló megközelítés azt jelenti, hogy addig vizsgálunk egy témát, amíg abból olyan (leginkább elméleti) sarokpontokat, alapvetéseket tudunk leszűrni, amelyek tovább általánosíthatók, és az eredmény felhasználhatóvá válik további kutatási és gyakorlati célokra. A szintetizáló módszer a mérnöki szemléletet is belevisz a kutatásba. Ebben az esetben kisebb megoldásokat felhasználva egy olyan nagy, komplex eredményt hozunk létre, amely értékét tekintve több mint a kis megoldások értékének összessége.

Először tekintsük át azokat a módszereket, amelyek inkább az *analizáló* kategóriába sorolhatók.

[17] egy olyan technikát mutat be, amely formális bizonyítását teszi lehetővé annak, hogy egy hozzáférési kérés megfelel-e egy hozzáférés-vezérlési házirendnek. Mindezt olyan környezetekben, ahol formális logikai definíciók határozzák meg a szabályokat. A felhasználói jogosultságokat egy elosztott rendszerbeli komponens szolgáltatja. A megoldás minimalizálni kívánja a hozzáférés jogosságát ellenőrző bizonyítás lépéseinek számát. A központi, centralizált adatbázis gondolata több szerzőnél is megjelent: [29].

Ha egy szolgáltatás több konkurens kérést szolgál ki, akkor sokszor szükség van arra, hogy kezelni tudjuk az összekapcsolódó, korreláló kéréseket, vagy válaszokat. Azok a tulajdonságok, amelyek lehetővé teszik a korrelációk kezelését, a szabványok és eszközök részei, azonban ezek kifejező ereje sokszor nem elégséges. [16] megmutatja ezeket a hiányosságokat leginkább a WS-Addressing [86] és a BPEL [44] tekintetében, majd pedig tárgyalja, hogy miként lehetne további korrelációs eseteket támogatni ezekkel az eszközökkel.

Az analizáló és a szintetizáló módszertanok határán állunk most. Nézzünk egy ide kapcsolódó kutatási eredményt! Amikor humán munkafolyamatok vezérléséről beszélünk, meg kell kötnünk azt, hogy ezek lépésit milyen felhasználó hajthatja végre. A [25] is ezt a problémát veti fel és keres megoldást rá. Azon felül, hogy megmondható, hogy milyen szerepkör tagjai hajthatnak végre egy műveletet, sokszor az is elvárás, hogy két különböző lépést ugyanaz a

felhasználó hajtson végre. Szintén előfordulhat ennek ellenkezője, amikor azt akarjuk biztosítani, hogy két lépést két különböző felhasználó hajthat csak végre. Sokszor elvárás az is egy humán munkafolyamatban, hogy a végrehajtási jogosultságok a felhasználók szervezeten belüli alárendeltségi viszonyai alapján változzanak (pl. a beosztott munkáját a főnök hagyhatja csak jóvá.) Egy adott művelet végrehajtásának darabszáma is korlátozható. Ennek ellenőrzésére készített algoritmikus megoldást a szerző.

A következőkben a szintetizáló módszerekre térek át. Az üzleti folyamatok modelljéből leszűrhetők a használati esetek, azonban ez legtöbbször nem elégséges ahhoz, hogy a megrendelő igényeinek megfelelő alkalmazást készíthessünk, ugyanis a munkafolyamatok a használati esetekből nem szűrhetők le 100 %-os biztonsággal. Léteznek olyan megoldások [31], amelyek az üzleti folyamat és a rendszermodellezés, rendszertervezés közé helyeznek egy olyan modellezési lépést, amely az üzleti folyamatokban végzett manuális lépéseket, az informatikai rendszer által automatikusan elvégezett műveleteket és a munkafolyamatok lépéseit elválasztja egymástól. Egyik megoldást erre a problémára a Work Analysis Refinement Modelling (WARM – Munka Analízis Finomító Modellezés) [74] nyújt. Ennél továbbmegy a szerző. A munkafolyamatok állomásaihoz legtöbbször Role Based Access Control (RBAC – Szerepkör alapú hozzáférés vezérlés) [70] alapján adunk jogosultságokat. A *(r, t, p)* hármas határozza meg azt, hogy az *r* szerepkörbe tartozó felhasználók a *p* feltétel esetében a *t* munkafolyamat lépést végrehajthatja. A WARM módszer egy továbbfejlesztése segítségével az üzleti folyamat modellből nemcsak a munkafolyamatok szűrhetők le, hanem a munkafolyamat lépéseihez köthető jogosultságok is.

Az elosztott alkalmazások kapcsoló interfészeit jelenleg leginkább Java-ban vagy .NET (pl. C# nyelven) környezetben készítjük el, majd webszolgáltatásként [79] esetleg más szabványos protokollon publikáljuk. A webszolgáltatások szabványos leíró nyelve a WSDL (Web Service Description Language – webszolgáltatás leíró nyelv). A WSDL nem tér ki magára az üzleti folyamatra, csak az interfészt definiálja. A BPEL (Business Process Execution Language – üzleti folyamatok végrehajtó/végrehajtható nyelve) [78][73] egy szabványos nyelv, amely már magát az üzleti folyamatot is leírja. A nyelv XML alapú, azonban csak egyszerű programozási nyelvi elemeket tartalmaz, ezért csak más programozási nyelvvel integrálva használható komolyabb célokra. A BPEL elődje a BPML, amely csupán a modellezést segítette. A BPEL nem rendelkezik magas szintű biztonsági elemekkel.

Összefoglalva elmondható, hogy a BPEL egy DSL (Domain Specific

Language – Szakterület specifikus nyelv), mert az egy konkrét feladatra kifejlesztett célnyelv. Mivel a (WS-)BPEL nincs felkészítve magas rendelkezésre állást megkövetelő, konkurens rendszerek kezelésére, léteznek más erre a célra szakosodott nyelvek is, pl. a GPSL [24]. Elmondható, hogy egyik sem rendelkezik olyan biztonsági elvárásokkal, amelyek megfelelnének a modern igényeknek.

4.4 Az alapgondolatok felvetése

A 4.3-es fejezetben kiemeltem néhány fontosabb cikket, amely az elosztott rendszerek biztonsági kérdéseivel kapcsolatban látott napvilágot. Az ezekből leszűrhető módszereken kívül, amivel a kutató dolgozott, más besorolási logika alapján is csoportosíthatjuk a munkákat. Ezek a következők:

1. Elosztott rendszerek publikus műveleteinek jogosultságairól
2. Munkafolyamatok modellezéséről
3. Munkafolyamatok lépéseiről és ezekhez kapcsolódó jogosultságokról

Jelen dolgozatnak nem célja a második pont témakörének tárgyalása.

A szerzők legtöbbször egymástól különálló problémakör elemeiként tárgyalják az elosztott rendszerek publikus műveleteinek jogosultságait és a munkafolyamatok lépéseinek jogosultságait, holott ez a kettő legtöbbször együtt jár. Ezt következetesen végiggondolva a következő állításokat tehetjük:

1. Ritka az az elosztott alkalmazás, ahol az üzleti homlokzaton (publikus interfészen) publikált műveletek tetszőleges sorrendben, feltételek nélkül meghívhatók (hívások sorrendjének meghatározására kellene módszer). A külső, egymás után történő behívásokat végző rendszer csak nem-humán munkafolyamat által vezérelt módon hajthatja végre a műveleteket.
2. Nem jellemző az a több szereplőt foglalkoztató humán munkafolyamat, amely nem elosztott alkalmazás-architektúrában kerül publikálásra.

A problémát ott látom, hogy szigetrendszerként modellezik a munkafolyamatokat és az üzleti szolgáltatások jogosultságkezelését, holott ezek szorosan integrálhatók, összekapcsolhatók lehetnének. Ennek megfelelően olyan további biztonsági kérések integrációját javaslom a következő fejezetekben, amely egy komplex informatikai rendszer működését, biztonságát, architektúráját, megérthetőségét tekintve továbblépés lehet.

A következő fejezetben valós életből vett esettanulmányokkal alátámasztva megismertetem az olvasót a munkafolyamatok és a publikus interfészek összeházasításának egy olyan általam kidolgozott módszerével [8], amely

előremutató a modern programfejlesztési paradigmákat tekintve, illetve egyszerűsíti, konszolidálja a munkafolyamatok publikus interfészen keresztül való kezelését.

4.5 Módszer a munkafolyamatok és az elosztott alkalmazások publikus interfészének integrációjára

Mielőtt a formális modellt szemléltetném, előkészítésképpen bevezetem a munkafolyamatok leírására felhasznált EDFSM fogalmát (4.5.1), valamint esettanulmányokat mutatok (4.5.2).

4.5.1 Az EDFSM definíciója

A munkafolyamatok leírásakor használt állapotdiagramokat véges automatának, véges állapotgépnek (FSM – Finite State Machine) [43] is tekinthetjük. A véges állapotgépek egy csoportja a determinisztikus állapotgép (DFSM), amely azt a megszorítást is tartalmazza, hogy minden (*állapot, bemeneti érték*) párra pontosan egy állapotátmenetet definiálunk egy másik állapothoz. Esetünkben azonban nem bemeneti értékekkel, illetve karaktereket tartalmazó ABC-vel dolgozunk, hanem eseményekkel (műveletekkel), amelyek az állapotátmenetet indukálják. (A továbbiakban az esemény, állapotátmenet illetve művelet fogalmakat egymással felcserélhető módon alkalmazom.) Az események, az ABC elemeiként manifesztálódnak. Ennek megfelelően eltérek a klasszikus állapotgép definíciótól és bevezetem az eseményvezérelt determinisztikus véges állapotgép (EDFSM – Event-driven Deterministic Finite State Machine) fogalmát. A valós életben nem feltétlenül visz át egy állapotból egy esemény egy következő állapotba, mivel ha nem teljesül valamilyen, az átmenethez szükséges előfeltétel, akkor az átmenet nem történik meg. Ezek alapján a következő definíciót adom:

$$\mathrm{EDFSM} \stackrel{\text{def}}{=} (\mathrm{E}, \mathrm{S}, \mathrm{s}_0, \mathrm{P}, \delta, \mathrm{F})$$

ahol

1. E az események véges, nem üres halmaza, amelyeket az EDFSM elfogad (ABC)
2. S egy véges nem üres állapothalmaz
3. s_0 kezdőállapot, ahol $s_0 \in S$
4. P az átmenetek, események előfeltételeinek véges halmaza
5. $\delta: S \times E \times P \rightarrow S$ állapotátmeneteket definiáló függvény
6. F véges, nem üres állapothalmaz, ahol $F \subset S$

Ha klasszikus állapotgépről beszélnénk, akkor azt definiálnánk, hogy milyen feltételek mellett fogadja el a gép ABC-je feletti szöveget az állapotgép. Az EDFSM esetében a definíció arra vonatkozik, hogy milyen állapotátmenet illetve művelet sorozat hatására hajtható végre egy üzleti folyamat az állapotgép felhasználásával. A következőt mondjuk: Legyen *edfsm* egy olyan EDFSM, ahol $M=e_0,e_1,\ldots,e_{n-1}$ egy állapotátmenet, azaz művelet lista E felett. Gyakorlatilag ez az üzleti folyamat során végrehajtott műveletek sorozata. $P=p_0,p_1,\ldots,p_{n-1}$ az azonos indexen található műveletek előfeltételeinek sorozata. Az *edfsm* segítségével levezényelhető az M üzleti folyamat, ha az $r_0,r_1,\ldots,r_n$ állapotok ($r \in S$) a következő feltételnek megfelelnek:

1. $r_0 = s_0$
2. $\forall i \in [1 \ldots n-1] : r_{i+1} = \delta(r_i,\ e_i\ p_i)$
3. $r_n \in F$

Informálisan fogalmazva a fenti három pont meghatározza azt, hogy mi a kezdőállapot, hogy pontosan milyen állapotátmenetek hajthatók végre, valamint azt, hogy mi a végállapot.

Az EDFSM-ben alkalmazott előfeltételhez kötött állapotátmenet ötlete nem teljesen új, ugyanis az UML [63] állapotdiagramja szintén alkalmazza ezt a koncepciót. Ahhoz, hogy az EDFSM DFSM-ként legyen értelmezhető, a következő állítást teszem:

Állítás: Minden EDFSM egyértelműen megfeleltethető egy DFSM-nek úgy, hogy a DFSM tulajdonságai nem sérülnek meg.

Bizonyítás:

Először megmutatom, hogy lehet egy EDFSM-et DFSM-re visszavezetni.

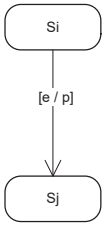
Az EDFSM a DFSM-hez képest a következő eltéréseket mutatja:

1. Az ABC elemei itt események és nem karakterek
2. Minden állapotátmenethez egy előfeltétel adható. Akkor és csak akkor megy végbe az esemény által indukált állapotátmenet, ha az előfeltétel igazra értékelődik ki.

Az első pontban vázolt ABC-beli különbség kezelésére minden EDFSM-ben értelmezett eseményhez egy DFSM-ben értelmezett karaktert feleltetünk meg. Az EDFSM alapján készítsünk egy olyan DFSM-et, amelynek állapotai (kezdő és végállapotokat is beleértve) megegyeznek. Az EDFSM állapotátmenetei helyett pedig a DFSM megfelelő karaktereit értjük ($e_i \leftrightarrow c_i$).

Tekintsünk egy olyan EDFSM-beli állapotátmenetet, ahol az s_i állapotból az s_j állapotba az *e* esemény visz át akkor, ha a *p* előfeltétel teljesül.

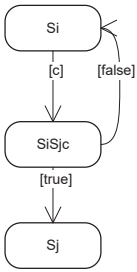
Ezt a következő ábra szemlélteti:



6. ábra: EDFSM-beli állapotátmenet

Az *e* eseményhez egy *c* karaktert feleltetek meg a DFSM-ben. A *p* logikai feltétel igazra (*true*) vagy pedig hamisra (*false*) értékelődhet ki. Miután egy olyan *c* karakter érkezik bemenetként, amely az EDFSM s_i állapotában egy legális *e* eseményhez tartozik, akkor egy *true* vagy egy *false* karakter is bekerül a bemeneti szövegbe. Az ABC-t ki kell egészítenünk a *true* és a *false* karakterekkel.

Tekintsük a következő ábrát, amely előző ábrán jelzett átmenet megfelelőjét mutatja:



7. ábra: DFSM-ben értelmezett állapotátmenet

A lehetséges eseteket a bemeneti értékre vonatkozóan a következő táblázatban foglalom össze:

Eset	EDFSM	DFSM
legális esemény, igaz p	…, e / p, …	…, c, true, …
legális esemény, hamis p	…, e / nem p, …	…, c, false, …
illegális esemény, nincs p	…, e2, …	…, c2, …

2. táblázat: EDFSM - DFSM megfeleltetési táblázat

Azaz megfeleltetés a bemeneti esemény-előfeltétel párokból indukált karaktersorozatok felhasználásával történik.

A fordított irány triviális az előfeltételek igazzá tételével.

4.5.2 Esettanulmányok

Az előzőekben megállapítottam, hogy egy publikus interfész műveletei általában nem tetszőleges sorrendben hívhatók meg, valamint, hogy a több szereplős humán munkafolyamatok általában elosztott alkalmazás-architektúrában kerülnek publikálásra.

Először arra az esetre mutatok példát, amikor a kliens oldalon egy gép áll, tehát nem ember végzi a műveleteket. Az „A" és a „B" művelet között akkor szokott előfordulni sorrendi kötöttség, ha az „A" művelet a „B" művelet részére valamilyen előkészítő lépéseket végez (pl. „A" inicializál valamit) illetve ha az „A olyan adatot állít elő, amire „B"-nek szüksége van.

Egy ilyen interfészt az Igazságügyi és Rendészeti Minisztérium (IRM) számára elkészített Elektronikus Beszámoló rendszer [97] használ fel előtétrendszerként. A rendszer létrehozásában tervezőként és vezető fejlesztőként vettem részt a kivitelező cég részéről.

Az Elektronikus Beszámoló (röviden e-Beszámoló) beadására kötelezett cégek az e-Beszámolót 2009. május 1-től csak és kizárólag elektronikus úton adhatják be az Ügyfélkapun [107] keresztül, amelyek a Hivatali Kapuhoz kerülnek. A Hivatali Kapun (lásd később) keresztül az IRM rendszere letölti az ügyfelek által feltöltött e-beszámolókat, automatikusan feldolgozza, ellenőrzi, majd pedig visszajelez a sikeres/sikertelen letöltésről az ügyfél számára. A visszajelzés szövegét az ügyfél alapértelmezés szerint az Értesítési Tárhelyén olvashatja az Ügyfélkapu felhasználói felületén. A helyesnek talált e-beszámolók az előbb említett webes felületen kereshetők és ismerhetők meg ingyenesen. A helyes e-beszámolók űrlap részét a rendszer átadja az APEH részére.

A Hivatali Kapu (HKP) gyakorlatilag egy postafiók szolgáltatást ad a kapcsolódó szervezeteknek, amin keresztül az ügyfelekkel (számukra láthatatlan módon) a kommunikáció történik. Ezt a webszolgáltatás interfészt és még sok minden mást a Közigazgatási Informatikai Bizottság 21. számú ajánlása [77] tartalmazza.

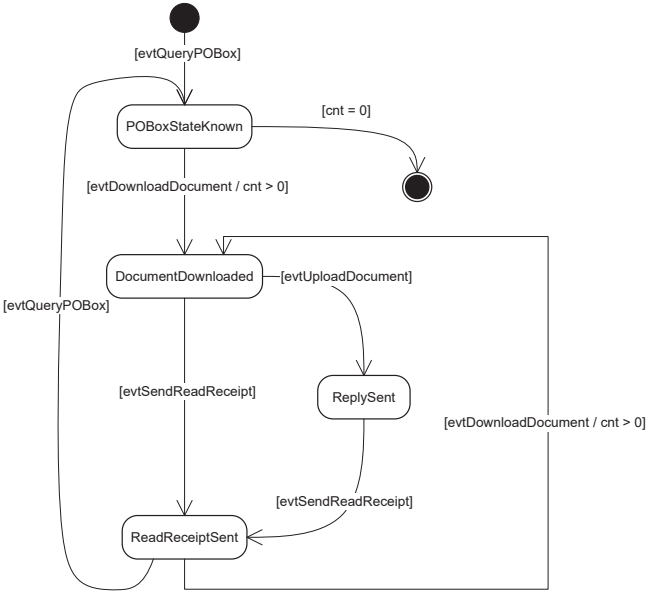
A következő HKP műveletek meghívása szükséges:

1. *Postafiók státusz*ának *lekérdez*ése, hogy tudjuk hány dokumentum érkezett az ügyfelektől.
2. *Dokumentum letöltése* funkció, amelynek segítségével az ügyfelektől

beérkezett dokumentumokat tölthetjük le.

3. *Dokumentum feltöltése* funkció, amelynek segítségével válaszüzenetet küldhetünk a felhasználónak a feldolgozás eredményéről. A gyakorlatban célszerű egy PDF formátumú dokumentumot küldeni, ahol szöveges formában értesül az ügyfél a feldolgozás eredményéről.
4. *Olvasási visszaigazolás* küldése, amely tudatja a HKP-vel, hogy a dokumentumot sikeresen letöltöttük és feldolgoztuk, azaz törölhető a postafiókból.

A *Postafiók státusz lekérdezés* művelet meghívása minden esetben a legelső kell, hogy legyen, különben nem tudjuk, hogy hány dokumentum tölthető le a postafiókból. Bejövő dokumentum esetében a *Dokumentum letöltése* funkció következhet. A válaszüzenet küldése a *Dokumentum feltöltése* funkcióval egy opcionális lépés, azonban a feldolgozást mindenképpen egy *Olvasási Visszaigazolás* műveletnek kell zárnia. Ezután az előző *Postafiók státusz lekérdezés* művelet alapján tudhatjuk, hogy még van-e letölthető dokumentum. Ha igen, akkor letöltünk egy újabb dokumentumot, ellenkező esetben pedig újra lekérdezzük a postafiók státuszt. A leírtakat a következő állapotdiagram szemlélteti:



8. ábra: HKP munkafolyamat

A *cnt* érték jelzi az új dokumentumok számát, amelyet a *PostafiókLekérdezés* művelet hatására ismerünk meg. Az egyszerűség kedvéért az ábrán nem jelöltem, de nyilvánvaló, hogy a *cnt* értéke minden dokumentum letöltése után eggyel csökken.

A HKP működését szemléltető diagram is leírható egy EDFSM-el.

Az állapotokat a következő rövidített formában jelölöm:

- S = StartState (Kezdőállapot)
- PSK = POBoxStateKnown (Postafiók státusz ismert)
- DL = DocumentDownloaded (Dokumentum letöltve)
- RS = ReplySent (Válasz üzenet elküldve)
- RRS = ReadReceiptSent (Olvasási visszaigazolás elküldve)
- E = EndState (Végállapot)

A következő táblázat szemlélteti, hogy milyen állapotból milyen esemény hatására, milyen feltételekkel jutunk el egy rákövetkező állapotba:

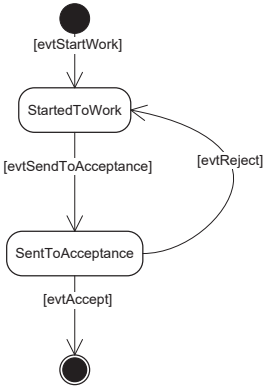
Esemény **Állapot**	**evtQuery-POBox**	**evtDownload Document**	**evtUpload-Document**	**evtSendRead-Receipt**	**evtStop**
S	PSK	-	-	-	-
PSK	-	cnt > 0 → DL	-	-	E
DL	-	-	RS	RRS	-
RS	-	-	-	RRS	-
RRS	PSK	cnt > 0 → DL	-	-	-
E	-	-	-	-	-

3. táblázat: HKP működése

A *cnt > 0* esetet mutatja a táblázat, ebben az esetben léphet a munkafolyamat a letöltő műveletre. Az állapotdiagram szerint, ha nincs új dokumentum (*cnt = 0*), akkor végállapotba kerülünk, azonban ez nem kapcsolódik konkrét eseményhez, mivel egy művelet kimeneti értékétől függ. A δ függvény a táblázat alapján egyértelműen felírható (a későbbiekben fogom megmutatni).

Most nézzük a másik és egyben gyakoribb esetet! Mint már említettem a humán munkafolyamatokat legtöbbször elosztott alkalmazások formájában valósítjuk meg. Egy tipikus és egyszerű példa erre az elfogadási munkafolyamat. Lényege az, hogy egy szervezeten belüli beosztott elkészít valamilyen feladatot,

amelyet egy informatikai rendszer segítségével eljuttat a főnökéhez, aki elfogadhatja vagy elutasíthatja beosztottjának munkáját. Elutasítás esetében a beosztott tovább dolgozik, majd újra megpróbálja elfogadtatni a munkát. Minden egyes művelet, más szóval esemény bekövetkezését emberek vezénylik egy-egy, a felhasználói felületen is látható nyomógomb segítségével. A következő állapotdiagram szemlélteti az elfogadási munkafolyamatot:



9. ábra: Elfogadási munkafolyamat

Az állapotokat a következő rövidített formában jelöljük:

- S = StartState (Kezdőállapot)
- SW =StartedToWork (Munkavégzés folyamatban)
- STA = SentToAcceptance (Elfogadás alatt)
- E = EndState (Végállapot)

A következő táblázat szintén szemlélteti, hogy milyen állapotból milyen esemény hatására, milyen feltételekkel jutunk el egy rákövetkező állapotba:

Esemény Állapot	evtStartWork	evtSendTo-Acceptance	evtAccept	evtReject
S	SW	-	-	-
SW	-	STA	-	-
STA	-	-	E	SW
E	-	-	-	-

4. táblázat: Elfogadási munkafolyamat működése

A δ függvény a táblázat alapján egyértelműen felírható.

4.5.3 A munkafolyamatok és az üzleti homlokzat integrálásának előkészítése

A 5. fejezetben többek között azt is ismertetem majd, hogy a 4.5.2-ben bemutatott példák miként implementálhatók egy általam kifejlesztett megoldás segítségével, amellyel bizonyítani kívánom a dolgozatban leírtak alkalmazhatóságát. Mindezek előtt azonban nézzük meg, hogy milyen klasszikus megoldási lehetőségek kínálkoznak a 4.4-ban bemutatott problémákra, amelyeket a 4.5.2-ben példákkal támasztottam alá!

A következő megoldásokat alkalmazzák az elosztott munkafolyamatok fejlesztői leggyakrabban:

1. Implementáljunk mindent kézzel! elve.
2. A munkafolyamat üzleti szolgáltatások mögé rejtése.
3. A munkafolyamat lépéseinek web szolgáltatásként való publikációja.

Az első megoldás azt ismerteti, amikor a fejlesztők a munkafolyamatok vezérlését natív módszerekkel, segédkönyvtár nélkül oldják meg, holott több könyvtár áll rendelkezésére, amelyek a munkafolyamatok hatékonyabb implementációját kívánják segíteni [109][100]. Ez gyakorlatilag azt jelenti, hogy az imperatív programozás eszközeit használják csak fel. Itt a munkafolyamat lépések elvégezhetőségének validálása az üzleti szolgáltatásokba kerül „bedrótozásra", amely csökkenti a kód olvashatóságát, karbantarthatóságát valamint több hibalehetőséget hordoz magában. A konkrét implementáció szintjén megfogalmazva az üzleti entitásba egy állapot (state) adatmező kerül felvitelre, amely alapján a bedrótozott logikai feltételek segítségével vezénylik a munkafolyamat lépéseit. Szerencsére ez az implementációs megoldás már kezd teret veszíteni, azonban még sok örökölt rendszerben látható, valamint egyszerű munkafolyamatok esetében még mindig használatos.

A második pont ennél egy fokkal jobb megoldást jelent (gyakorlatban ez a legelterjedtebb). Ebben az esetben az üzleti műveletek kódja valamilyen munkafolyamatok vezérlését támogató segédkönyvtárnak adja át a vezérlést. Ilyenkor a segédkönyvtár két különböző szerepkörben, működési módban létezhet:

2/1. A művelet elvégezhetőségét validálja

2/2. A művelet elvégezhetőségét validálja, valamint elvégzi a műveletet, amennyiben megengedett

Az 2/1. esetben előny, hogy a munkafolyamatban elvégezhető üzleti kód az üzleti műveletben marad, az elvégezhetőséget ellenőrző programkód pedig kikerül az üzleti kód mellől. Mivel a munkafolyamat lépések elvégezhetőségének

vizsgálata ortogonális műveletnek tekinthető, és így függetlenné válik a kódtól, ezért ez a megoldás célravezető lehet. Gyakorlatban azonban a segédkönyvtárak a 2/2. megoldást támogatják, amelyben a segédkönyvtár vezényli az elvégezhetőség vizsgálatát valamint magának a műveletnek az elvégzését is.

A harmadik megoldás valójában a 2/2 kiterjesztése oly módon, hogy az üzleti homlokzaton kipublikálja a munkafolyamat műveleteit. Ez leginkább szabványos webszolgáltatás interfészen keresztül történik.

Látható, hogy a leginkább objektumorientált megoldás a 2/1-es, ennek ellenére ezzel viszonylag ritkán találkozunk. A helyes szemléletmód természetesen mindig az, amely leginkább objektumorientált szemléletmódot sugall, ugyanis az így felépített modell áll legközelebb a való világban megtalálható fogalmakhoz. Maguk a műveletek illetve a műveletek elvégzésének validálása egymáshoz képest ortogonális koncepciók. Ezáltal kötelező a programot úgy felépíteni, hogy a validáló kódok között magas legyen a koherencia [63] foka, ugyan úgy, ahogy maguk a műveletek között is. A validációk és a műveletek között pedig kisebb koherencia kialakítása javasolt.

A 4.5 fejezet címe a publikus interfész és a munkafolyamatok integrálásáról beszél. Az eddig leírtak szerint ez még nem történt meg olyan szinten, mint ahogy ez elvárható lenne. Egy szerződést kell kiépíteni a kliens és a szerver között, amelyben rögzíteni szükséges a következőket:

1. milyen állapotban,
2. milyen művelet,
3. milyen előfeltételekkel

hajtható végre. Ezeket a feltételeket annak tudatában, hogy az állapot mindig ismert, a következőkké konvertáltam:

1. Milyen munkafolyamat kapcsolható az eseményhez?
2. Milyen előfeltételek fennállása esetében végezhető el az állapotátmenet?

Ezek tekinthetők az üzleti műveletek attribútumainak is. Ahhoz, hogy ezen attribútumokat jelölni tudjuk, bevezetek egy formalizmust.

4.5.4 A munkafolyamatok és az üzleti homlokzat formális integrálása

A munkafolyamatok és az üzleti homlokzat formális integrálását Bertrand Meyer által kifejlesztett és tökéletesített szerződésmodell [53] (Design by Contract, lásd 4.1.) ihlette. Az elv lényege, hogy a szoftverkomponensek egy precíz, ellenőrizhető interfészdefinícióval kell, hogy rendelkezzenek. Ezt a definíciót Meyer kiterjesztette úgy, hogy előfeltételek, utófeltételek valamint invariánsok is megadhatók a szerződésben. Ez a szerződés összekapcsolható az elkészült program osztályaival illetve műveleteivel, amelyeknek a szerződéssel való megfelelőségét ellenőrizhetjük.

Mivel ez egy jól bevált és kipróbált [54] elv, ezért dolgozatomban ezekre a koncepciókra építkezve egy elosztott alkalmazásokra kiterjesztett megoldást mutatok be. A formalizmus kidolgozásánál azt tartottam szem előtt, hogy abból minél könnyebben működő implementációt legyen készíthető.

Ahhoz, hogy megszorításokat lehessen megfogalmazni egy szerződéssel és a szerződés műveleteivel szemben, először magát a szerződést definiálom:

$$C \stackrel{\text{def}}{\equiv} \left(\{m_1, m_2, \dots, m_n\}, R_C, \{R_{m_1}, R_{m_2}, \dots, R_{m_n}\}, \text{edfsm}\right)$$

1. egyenlet: Szerződésdefiníció

A C szerződés tehát n db metódust definiál $(m_1, m_2, \dots, m_n)$, ahol $m_i \in M_C$ (M_C a C szerződés metódusainak halmaza). Az R_C jelzi a szerződés, szolgáltatás szintű megszorításokat, az $R_{m_1}, R_{m_2}, \dots, R_{m_n}$ pedig az $m_1, m_2, \dots, m_n$ metódusok számára adnak megszorításokat. Az *edfsm* jelzi azt az EDFSM példányt, amely a munkafolyamat formális definícióját adja.

A fenti definícióból ismeretlen az R_C és az R_{m_i} meghatározása. Az R_C-t a következőképpen definiálom:

$$R_c \stackrel{\text{def}}{\equiv} \left(\{t_{c,1}, t_{c,2}, \dots, t_{c,q}\}, \{id_{c,1}, id_{c,2}, \dots, id_{c,w}\}, \{net_{c,1}, net_{c,2}, \dots, net_{c,e}\}\right)$$

2. egyenlet: Szerződés szintű megszorítások

A definícióban szereplő *t* értékek a futás idejű hozzáférés-vezérlés kiterjesztéséhez használhatók (4.6. fejezet), az *id* értékek hívó identitás-korrelációhoz kapcsolódnak (4.7. fejezet), a *net* értékek pedig a hálózati biztonsághoz (4.8. fejezet).

Lássuk az R_{m_i} definícióját:

$$R_{m_i} \stackrel{\text{def}}{=} (\{t_{m_i,1}, t_{m_i,2}, \dots, t_{m_i,r_i}\},$$
$$\{(s_{m_i,1}, e_{m_i,1}, p_{m_i,1}), (s_{m_i,2}, e_{m_i,2}, p_{m_i,2}), \dots, (s_{m_i,y_i}, e_{m_i,y_i}, p_{m_i,y_i})\},$$
$$L(id_{m_i,1}, id_{m_i,2}, \dots, id_{m_i,u_i}), \{net_{m_i,1}, net_{m_i,2}, \dots, net_{m_i,o_i}\})$$

3. egyenlet: Metódus szintű megszorítások

Az $s_{m_i,j}$, az $e_{m_i,j}$ és a $p_{m_i,j}$ az EDFSM állapotai, eseményei illetve feltételei közül valók, tehát az EDFSM definícióját felhasználva a következő megkötés áll:

$$\forall i \epsilon [1 \dots n]: \forall j \epsilon [1 \dots y_i] \begin{cases} s_{m_i,j} \epsilon S \\ e_{m_i,j} \epsilon E \\ (s_{m_i,j}, e_{m_i,j}, p_{m_i,j}) \epsilon D_\delta \end{cases}$$

Az $(s_{m_i,j}, p_{m_i,j})$ párok az m_i metódus megszorításai között találhatók, amely gyakorlatilag azt jelenti, hogy a metódus akkor és csak akkor futtatható, ha az EDFSM definíciója szerinti $s_{m_i,j}$ állapotban van az állapotgép, valamint teljesül a $p_{m_i,j}$ feltétel. Az m_i futtatása az $e_{m_i,j}$ eseményt indukálja valamely j-re. Azokat a lehetséges állapotokat, ahol az m_i metódus futtatható $s_{m_i,j}$ jelöléssel látjuk el. Ezekben az $s_{m_i,j}$ állapotokban értelmezett az $e_{m_i,j}$ esemény valamely $p_{m_i,j}$ feltétel mellett. Azaz máshogyan fogalmazva, akkor és csak futtatható az m_i metódus, ha a δ függvény értelmezési tartományában megtalálható az $(s_{m_i,j}, e_{m_i,j}, p_{m_i,j})$ hármas.

Amennyiben az $(s_{m_i,j}, e_{m_i,j}, p_{m_i,j})$ hármasokra fennáll a $\exists k \neq l \in [1 \dots y_i]: s_{m_i,l} = s_{m_i,k}$ feltétel, akkor explicit módon abban az esetben dönthető el, hogy az $e_{m_i,l}$ illetve az $e_{m_i,k}$ közül melyik esemény váltódik ki, ha $p_{m_i,k} \wedge p_{m_i,l} =$ hamis. Ez azt jelenti, hogy ha egy metódusnál olyan események vannak megadva, amelyek ugyanabban az állapotban értelmezettek, akkor explicit módon csak abban az esetben határozható meg, hogy melyik esemény hajtódik végre (ennek hatásaként melyik állapotba jutunk), ha a futás *p* feltételei kizárják egymást.

Gyakorlatilag ez azt jelenti, hogy a $(s_{m_i,j}, p_{m_i,j})$ párok esetében, ahol $\exists k \neq l \in [1 \dots y_i]: s_{m_i,l} = s_{m_i,k}$ valamint a $p_{m_i,k} \wedge p_{m_i,l} =$ hamis feltétel teljesül, az $e_{m_i,j}$ kihagyható a megszorítások közül, ugyanis ebben az esetben egyértelmű, hogy melyik állapotátmenet következik be.

A fenti feltételek tejlesülése esetében valamint a $p_{m_i,k} \wedge p_{m_i,l} =$ igaz teljesülése esetében nem határozható meg egyértelműen, hogy melyik eseményt

indukálja a metódushívás.

Ennek az anomáliának az elkerülése érdekében azt a megszorítást teszem, hogy az $(s_{m_i,j}, e_{m_i,j}, p_{m_i,j})$ hármasok esetében $\forall k \neq l \in [1 \dots y_i]: e_{m_i,l} = e_{m_i,k}$ feltételnek teljesülnie kell.

Ez a megszorítás teljes mértékben tükrözi a gyakorlatot, ugyanis egy üzleti szolgáltatás interfészen publikált metódus minden esetben ugyanazt a munkafolyamat állapotátmenetet szokta indukálni. A különbség legfeljebb annyi, hogy a különböző állapotokban különböző feltételek fennállása esetében történik meg az állapotátmenet. Pontosan ezt vezettem le a fentiekben. Ennek megfelelően a következőképpen egyszerűsíthető a metódusokra vonatkozó megszorítás definíciója:

$$R_{m_i} \stackrel{\text{def}}{=} (\{t_{m_i,1}, t_{m_i,2}, \dots, t_{m_i,r_i}\}, \{e_{m_i}, \{(s_{m_i,1}, p_{m_i,1}), (s_{m_i,2}, p_{m_i,2}), \dots,$$
$$(s_{m_i,y_i}, p_{m_i,y_i})\}\}, L(id_{m_i,1}, id_{m_i,2}, \dots, id_{m_i,u_i}), \{net_{m_i,1}, net_{m_i,2}, \dots, net_{m_i,o_i}\})$$

4. egyenlet: Metódus szintű megszorítások (eseménykiemelt)

Ezt a racionalizált definíciót alapul véve, ahol egyértelmű összerendelés áll fenn a metódus és az általa indukált munkafolyamat esemény között, tovább egyszerűsíthető az R_{m_i} megszorítások definíciója.

Az *edfsm* egyértelműen meghatározza a δ függvény által, hogy az e_{m_i} esemény milyen $s_{m_i,j}$ állapotokban milyen $p_{m_i,j}$ feltételek mellett hajtható végre. Ebből következően az $(s_{m_i,j}, p_{m_i,j})$ párok elhagyhatók lehetnek az R_{m_i} definíciójából. Ha azonban a való élet problémáira is tekintünk, akkor nyilvánvalóvá válik, hogy a $p_{m_i,j}$ feltételek olyan tulajdonságokat is figyelembe vehetnek, amelyek a munkafolyamat szintjén nem állnak rendelkezésre, csak az üzleti szolgáltatás szinten, ezért célszerű kiterjeszteni $p_{m_i,j}$ állapot-átmeneti feltételek szerepét. Ez annyit jelent, hogy nemcsak a munkafolyamatból származó feltételek vizsgálhatók, hanem az üzleti logika vagy üzleti entitások szintjén elérhető paraméterek is kiértékelhetők itt. Erre kitűnő példa, amikor egy közbeszerzési eljárásban az értékhatártól függően más szervezethez kell jóváhagyásra továbbítani a beszerzési igényt. Az értékhatár még ismert lehet a munkafolyamatban, azonban a konkrét entitásban definiált érték már nem, ezért ezt a munkafolyamaton kívülről kell venni. Ez természetesen csak egy implementáció szintű kérdés, a formalizmus szintjén nem szignifikáns.

A $(s_{m_i,j}, p_{m_i,j})$ párok ilyen szigorú felsorolása értelmetlen amennyiben $\forall k \neq l \in [1 \dots y_i]: p_{m_i,l} = p_{m_i,k}$, ugyanis ebben az esetben tetszőleges, az *edfsm* által megengedett állapotban ugyanaz a feltétel él. Továbbmenve, amennyiben a

$p_{m_i,j}$ feltételek üresek, akkor a munkafolyamatra vonatkozóan nem szükséges az eseményen kívül további információt feltüntetni az R_{m_i}-ben.

Ezek alapján így módosul a metódus szintű megszorítások definíciója:

$$R_{m_i} \stackrel{\text{def}}{=} (\{t_{m_i,1}, t_{m_i,2}, \dots, t_{m_i,r_i}\},$$
$$\left\{ \begin{pmatrix} e_{m_i}, \\ (s_{m_i,1}, p_{m_i,1}), \dots, (s_{m_i,y_i}, p_{m_i,y_i}), \text{ha } \exists k \neq l \in [1 \dots y_i]: p_{m_i,l} \neq p_{m_i,k} \\ (*, p_{m_i}), \text{ha } \forall k \neq l \in [1 \dots y_i]: p_{m_i,l} = p_{m_i,k} \\ \emptyset, ha\ \forall k \in [1 \dots y_i]: p_{m_i,k} = \emptyset \end{pmatrix} \right\}$$
$$L(id_{m_i,1}, id_{m_i,2}, \dots, id_{m_i,u_i}), \{net_{m_i,1}, net_{m_i,2}, \dots, net_{m_i,o_i}\})$$

5. egyenlet: Metódus szintű megszorítások (racionalizált)

A metódus szintű megszorítások definíciója (R_{m_i}) az eredeti verzióhoz képest komplexebbé vált, amint az ipari környezetben is előforduló használati esetek alapján megkerestem az egyszerűbb aleseteket és ezeket egy integrált formulával fejeztem ki. Mivel a dolgozatban ismertetett formalizmus nem öncélú, hanem a későbbi implementációhoz kíván segítséget nyújtani, ezért a korábban ismertetett esettanulmányok (4.5.2) formalizálásával (4.10) illetve a formalizmus alapján készített implementáció (5.3) megmutatásával kívánom a formalizmus előnyét felvázolni.

4.6 A klasszikus futás idejű hozzáférés-vezérlés kiterjesztése elosztott alkalmazásokra

A következő megállapítást már 4.1-ben is megtettem, de most felelevenítem, hogy kihangsúlyozzam a lényegét: amennyiben csökkentjük (és egyben racionalizáljuk) azokat az interfészeket, amelyeken keresztül a szoftver komponensek kommunikálhatnak egymással, akkor növeljük a szoftver minőségét és csökkentjük a fejlesztési költséget. A láthatóság vizsgálat vagy hozzáférés-vezérlés minden modern programozási nyelvnek illetve futtatókörnyezetnek része. Ezek a megoldások maghatározzák a szoftver komponensek felelősségi köreit, implementációs és biztonsági házirendeket (policy) definiálnak.

Az m_i metódusra vonatkozó formális megszorítások közül a $t_{m_i,1}, t_{m_i,2}, \dots, t_{m_i,r_i}$ jelzi azoknak a típusoknak körét, amelyekből történő m_i metódushívás engedélyezett. A szerződéshez tartozó megszorítások közül a $t_{c_1}, t_{c_2}, \dots, t_{c_q}$ vonatkozik ide. Akkor és csak akkor engedélyezett az m_i metódus meghívása egy t hívó típusból, ha az mind a metódus, mind pedig a szolgáltatás szintű engedélyezett típuslista eleme.

4.6.1 Az elosztott hozzáférés-vezérlés néhány használati esete

Egy szolgáltatás orientált elosztott alkalmazás esetében első gondolatunk az lehetne, hogy teljesen felesleges elosztott alkalmazások esetében hozzáférés-vezérlésről beszélni. Ezzel a lehetőséggel azt tudnánk meghatározni, hogy melyek azok a külső komponensek, illetve komponensekben lakozó osztályok, amelyekből a szolgáltatásunk összes illetve bizonyos metódusai meghívhatók.

Kézenfekvő megoldás az, hogy a szerver oldali szolgáltatásokhoz egy kliens oldali adaptert, proxy-t [37] készítsünk és kikényszerítsük azt, hogy csak ezeken az osztályokon keresztül legyen elérhető a szerver oldali szolgáltatás.

Empirikus tapasztalatok alapján megállapítható, hogy a szolgáltatási logikát az ismeri legjobban, aki készítette. Ebből következően, egy kliens oldali adaptert, proxyt az a szoftverfejlesztő tud legnagyobb szakértelemmel elkészíteni, aki magát a szerver oldali logikát is készítette. Ezek az osztályok, ún. kliens API-ként adhatók közre.

A proxy segítségével nemcsak a hívás szakszerűsége kényszeríthető ki, hanem az egész rendszer teljesítménye, performanciája növelhető.

Az elosztott alkalmazások hatékonyságának növelésére két bevált megoldás létezik:

1. A távoli metódusok hívása előtt a hívási paraméterek kliens oldalon való ellenőrzése.
2. Állandó adatok kliens oldalon történő gyorsító tárazása (*cache*-elése).

Ezek a megoldások természetesen egyedül vagy együtt is alkalmazhatók.

A teljesítmény növelésének egyik eszköze tehát az, hogy akkor és csak akkor fordulunk a szolgáltatáshoz, ha a hívó oldalon rendelkezésre álló információk alapján a hívási paramétereket ellenőriztük és helyesnek találtuk. Legtöbbször a következőket ellenőrizzük a hívó oldalon:

1. A kötelező paraméterek meghatározottsága.
2. A szöveges paraméterek hossza a megfelelő intervallumban legyen.
3. A numerikus paraméterek értéke a megfelelő intervallumban legyen.
4. A különböző paraméterek közötti összefüggések fennállása.

A másik megoldás tehát a gyorsító tárazás, azaz a cache alkalmazása. Ennek segítségével azokat az adatokat, amelyek csak ritkán változnak (pl. Magyarország megyéi vagy települései, cég partnereinek listája, stb.) a hívó oldalon a memóriában eltároljuk, majd híváskor a szerveroldali hívás helyett a cache adatait használjuk fel újra. Kicsit előremutatóbb, de ugyanezen az elven működő

megoldás az, hogy amennyiben tudjuk azt, hogy a hívó mikor kért le valamilyen adathalmazt, valamint a szerver oldalon eltároljuk az entitások legutóbbi változtatásának időpontját, akkor lehetőségünk van rá, illetve elegendő csak a változásokat elkérni a szervertől. Felhívom a figyelmet, hogy a kommunikációs, adatátviteli, adatbetöltési költség csökken, azonban az algoritmikus komplexitás növekszik.

A fentiek alapján ennek a kliens oldali proxynak feladata lehet a fent tárgyalt adatvalidáció illetve gyorsító tárazás ellátása. A cél tehát a hívó rákényszerítése arra, hogy ezen a proxyn keresztül kommunikáljon. Ez csak úgy oldható meg, ha a szolgáltatás oldalon ellenőrizzük a hívó komponenst és típust.

4.7 A hívó identitás-korreláció kikényszerítése

A 4.5 és a 4.6 fejezetben szóltam a munkafolyamatok és a hozzáférés-vezérlés elosztott alkalmazásokkal történő integrációjáról.

Tudjuk, hogy a mai elosztott rendszerek implementációját támogató komponensek legtöbbször szerepkör illetve csoporttagság alapú jogosultságkezelést alkalmaznak a hívó felhasználó korlátozására. Ez gyakorlatilag azt jelenti, hogy a szolgáltatás illetve a metódus szinten korlátozható az, hogy milyen felhasználók érhetik el a szolgáltatás metódusait.

Mint ahogy [25] is rávilágított, a munkafolyamatok különböző lépéseinek végrehajtója sokszor pl. ugyanaz vagy éppen különböző természetes személy kell, hogy legyen. Az itt található gondolatokat viszem tovább, fejtem ki bővebben.

Az általam kidolgozott koncepció összekapcsolja a szolgáltatás metódusait az EDFSM eseményeivel, azaz egy metódus mindig ugyanazt a munkafolyamat eseményt indukálja. Innentől kezdve akár még a munkafolyamatok esetében is szokásosnak mondható szerepkör alapú biztonsági mechanizmusok mellett metódusszinten határozható meg az is, hogy milyen korreláció áll fenn pl. két különböző metódus hívójának identitása között.

A szolgáltatás szintű $\mathrm{id}_{c_1}, \mathrm{id}_{c_2}, \dots, \mathrm{id}_{c_w}$ megszorítások a megszokott szerepkör, illetve felhasználói szintű megszorításokat definiálnak, azonban az m_i metódusra vonatkozó $\mathrm{id}_{m_i,1}, \mathrm{id}_{m_i,2}, \dots, \mathrm{id}_{m_i,u_i}$ megszorítások ezek mellett korrelációs szabályokat is definiálhatnak az m_i metódusra. Ebből kifolyólag a metódus szintű megszorításokat egy L logika kifejezésbe foglalom, amely a logikai kiértékelés eredményeként azoknak az elemeknek a halmazát adja vissza, amelyekre igaz az L-ben foglalt feltétel. A logikai kifejezés kiértékelése az alapvető logikai műveletekkel ($\wedge, \vee, \neg$) kapcsolja össze az operandusait.

Legyen MU = $((m,u)_{h1}, (m,u)_{h2}, \dots, (m,u)_{hn})$ a (metódus, hívó

felhasználó) párok sorozata. MU[a] jelöli az MU sorozat *a* indexén található párt. Legyen

MU :: last(a, m) = $MU[h] \mid h = \max(b \mid MU[b].m = m \wedge b < a)$,

azaz az MU :: last(a, m) függvény jelöli azt az MU-beli párt, ahol az *a* index előtt álló párok közül legutoljára az *m* metódus meghívása megtörtént.

A korrelációs szabályok a következők lehetnek, miközben a szabályok definiálása során tegyük fel, hogy MU[a].m = m_i, azaz az *a* indexen az m_i metódus meghívása történik:

1. $MU :: Same_{m_i}(m_j)$ fennállása esetében teljesülni kell a következőnek: Ha l_j = MU :: last(a, m_j) definiált, akkor l_j.u = MU[a].u. Informálisan fogalmazva az m_i metódus meghívásakor teljesülni kell annak a feltételnek, hogy az előzőleg meghívott legutolsó m_j metódust (ha volt ilyen) ugyanaz a felhasználó hívta, aki az m_i-t hívja most.
2. $MU :: SelfSame_{m_i} = MU :: Same_{m_i}(m_i)$. Informálisan fogalmazva az m_i metódus meghívásakor teljesülni kell annak a feltételnek, hogy az előző m_i metódushívást (ha volt ilyen) ugyanaz a felhasználó végezte, aki az aktuálisat.
3. $MU :: Diff_{m_i}(m_j)$. Ha l_j = MU :: last(a, m_j) definiált, akkor l_j.u <> MU[a].u. Azaz az m_i metódus meghívásakor teljesülni kell annak a feltételnek, hogy az előzőleg meghívott legutolsó m_j metódust (ha volt ilyen) más felhasználó hívta, mint aki az m_i-t hívja most.
4. $MU :: SelfDiff_{m_i} = MU :: Diff_{m_i}(m_i)$. Azaz az m_i metódus meghívásakor teljesülni kell annak a feltételnek, hogy az előzőleg meghívott legutolsó m_i metódust (ha volt ilyen) más felhasználó hívta, mint aki az aktuálisat.
5. $MU :: Subst_{m_i}(m_j)$. Ha l_j = MU :: last(a, m_j) definiált, akkor l_j.u helyettese MU[a].u. Azaz az m_i metódus meghívásakor teljesülni kell annak a feltételnek, hogy az előzőleg meghívott legutolsó m_j metódust (ha volt ilyen) hívó felhasználót az m_i-t hívó helyettesíti.
6. $MU :: SelfSubst_{m_i} = MU :: Diff_{m_i}(m_i)$. Azaz az m_i metódus meghívásakor teljesülni kell annak a feltételnek, hogy az előzőleg meghívott legutolsó m_i metódust (ha volt ilyen) hívó felhasználó helyettese az aktuális hívó.
7. $MU :: Sup_{m_i}(m_j)$. Ha l_j = MU :: last(a, m_j) definiált, akkor MU[a].u felettese l_j.u-nak. Azaz az m_i metódus meghívásakor teljesülni kell annak a feltételnek, hogy az előzőleg meghívott legutolsó m_j metódust (ha volt ilyen) futtató felhasználó felettese az m_i-t hívó.

8. $MU :: Inf_{m_i}(m_j)$. Ha l_j = MU :: last(a, m_j) definiált, akkor MU[a].u beosztottja l_j.u-nak. Azaz az m_i metódus meghívásakor teljesülni kell annak a feltételnek, hogy az előzőleg meghívott legutolsó m_j metódust (ha volt ilyen) futtató felhasználó beosztottja az m_i-t hívó.

4.8 A hálózati biztonság granularitásának finomítása

A szolgáltatásokat legtöbbször valamilyen alkalmazásszerver (pl. Microsoft IIS [105], IBM WebSphere [94], stb.) segítségével vagy valamilyen feladatra dedikált, folyamatosan futó folyamat (Windows Service, UNIX daemon, stb.) kontextusában publikáljuk.

Nyilvánvaló, hogy kötelező korlátozni azokat a hálózati szegmenseket, ahonnan ezek a szolgáltatáscsoportok, szolgáltatások hozzáférhetők. A komolyabb, integrált alkalmazásszerverek (pl. Microsoft IIS) tartalmaznak olyan beállítási lehetőségeket, ahol akár szolgáltatás szinten korlátozható azoknak az IP címeknek, hálózati szegmensek köre, ahonnan a szolgáltatások elérhetők.

Szintén megoldható az, hogy egy tűzfalat konfigurálunk fel olyan módon, hogy a szolgáltatásokhoz való hozzáférés a lehető legszűkebb hálózati szegmensekből legyen engedélyezett. A kevesebb tudással rendelkező tűzfalak csak port szinten képesek korlátozni a hozzáférést. Egy porton (pl. HTTP/HTTPS) több kipublikált szolgáltatás is futhat egyszerre. Ez azt jelenti, hogy e tűzfalak esetében olyan kliens is hálózati hozzáférést kaphat adott porton futó összes szolgáltatáshoz, amelynek csak egy vagy néhány szolgáltatáshoz szükséges hálózati elérési lehetőséget adni. Komolyabb tűzfalak (pl. Microsoft ISA Server [95]) szolgáltatás (URL) szinten képesek a korlátozást elvégezni.

Kézenfekvő integrált továbbfejlesztési lehetőségként kínálkozik a következő: ha tovább kívánjuk növelni a biztonsági szintet, valamint egy konkrét alkalmazás esetében ezen követelmény megoldható, illetve értelmezhető, akkor a hozzáférés vezérlési szabályokat tovább bővíthetjük. Mindez úgy valósítható meg, hogy szolgáltatás, illetve metódus szinten is megadjuk azokat az IP címeket, hálózati szegmenseket, ahonnan a művelet elérhető.

A szolgáltatás szintű hálózati megszorításokat a $\mathrm{net}_{c_1}, \mathrm{net}_{c_2}, \ldots, \mathrm{net}_{c_e}$ elemek adják, az m_i metódusra vonatkozó megszorításokat pedig a $\mathrm{net}_{m_i,1}, \mathrm{net}_{m_i,2}, \ldots, \mathrm{net}_{m_i,o_i}$ elemek.

Nem zárható ki annak a lehetősége sem, hogy a hívó indentitás-korrelációs megoldásomhoz hasonló megoldásról beszéljünk, azaz a különböző metódusok hívásának hálózati helye szerinti korrelációt is vizsgáljuk. Ezt a lehetőséget a 4.7-hoz képesti kevesebb újdonságtartalma miatt most nem ismertetem.

4.9 A legális metódushívás

Legyen H_m az az információhalmaz, amely az m metódus meghívásakor a szerver oldalnak szükséges ahhoz, hogy a hívás érvényességét ellenőrizze:

$$H_m = (m, t, id, net),$$

ahol

1. *m* a meghívott metódust jelzi,
2. *t* a kliens oldali hívást intéző osztályt jelzi,
3. *id* a hívó felhasználói identitást tartalmazza,
4. *net* a hívó hálózati azonosítóját (IP cím) hordozza.

A legális metódushívás formális definiálásához felhasználjuk a szolgáltatás szintű megszorítások (2. egyenlet)

$$\mathrm{R_c} \stackrel{\text{def}}{=} \left(\{\mathrm{t_{c,1}}, \mathrm{t_{c,2}}, \dots, \mathrm{t_{c,q}}\}, \{\mathrm{id_{c,1}}, \mathrm{id_{c,2}}, \dots, \mathrm{id_{c,w}}\}, \{\mathrm{net_{c,1}}, \mathrm{net_{c,2}}, \dots, \mathrm{net_{c,e}}\}\right),$$

illetve a metódus szintű megszorítások (5. egyenlet)

$$\mathrm{R_{m_i}} \stackrel{\text{def}}{=} \left(\{\mathrm{t_{m_i,1}}, \mathrm{t_{m_i,2}}, \dots, \mathrm{t_{m_i,r_i}}\}, \right.$$

$$e_{\mathrm{m_i}},$$

$$\left\{ \begin{pmatrix} (\mathrm{s_{m_i,1}}, \mathrm{p_{m_i,1}}), \dots, (\mathrm{s_{m_i,y_i}}, \mathrm{p_{m_i,y_i}}), \text{ha } \exists k \neq l \in [1 \dots \mathrm{y_i}]: \mathrm{p_{m_i,l}} \neq \mathrm{p_{m_i,k}} \\ (*, p_{\mathrm{m_i}}), \text{ha } \forall k \neq l \in [1 \dots \mathrm{y_i}]: \mathrm{p_{m_i,l}} = \mathrm{p_{m_i,k}} \\ \emptyset, ha\ \forall k \in [1 \dots \mathrm{y_i}]: \mathrm{p_{m_i,k}} = \emptyset \end{pmatrix} \right\}$$

$$\left. \mathrm{L}(\mathrm{id_{m_i,1}}, \mathrm{id_{m_i,2}}, \dots, \mathrm{id_{m_i,u_i}}), \{\mathrm{net_{m_i,1}}, \mathrm{net_{m_i,2}}, \dots, \mathrm{net_{m_i,o_i}}\}\right)$$

definíciójában alkalmazott jelöléseket.

A hívó kliens oldali osztályára, felhasználói identitására, hálózati azonosítójára vonatkozó feltételek validálását egyszerű halmazműveletek segítségével absztrahálom.

Az *eval(U)* függvény az U halmazban található elemeket, szabályokat értékeli ki úgy, hogy a kiértékelés eredményhalmazának legyen egyenlő az összes U-beli elem illetve U-beli szabályoknak megfelelő elem uniójával.

Legyen $eval^{\infty}(U) = \begin{cases} eval(U), & ha\ U \neq \emptyset \\ \bigcup_{\forall A \supset \emptyset} A, & ha\ U = \emptyset \end{cases}$, azaz nem üres halmaz esetében az eval(U)-t adja vissza üres halmaz esetében pedig az összes univerzumbeli elemet.

Ha a H_m információhalmazra igaz a következő, akkor legális a hívás:

1. $t \in eval(\mathrm{t_{c,1}}, \mathrm{t_{c,2}}, \dots, \mathrm{t_{c,q}}) \cap eval^{\infty}(\mathrm{t_{m,1}}, \mathrm{t_{m,2}}, \dots, \mathrm{t_{m,r}})$
2. $id \in eval(\mathrm{id_{c,1}}, \mathrm{id_{c,2}}, \dots, \mathrm{id_{c,w}}) \cap eval^{\infty}(L(\mathrm{id_{m,1}}, \mathrm{id_{m,2}}, \dots, \mathrm{id_{m,u}}))$
3. $net \in eval(\mathrm{net_{c,1}}, \mathrm{net_{c,2}}, \dots, \mathrm{net_{c,e}}) \cap eval^{\infty}(\mathrm{net_{m,1}}, \mathrm{net_{m,2}}, \dots, \mathrm{net_{m,o}})$

4. Legyen *s* az *edfsm* munkafolyamat aktuális híváskori állapota. Ekkor teljesülni kell a következőnek: $y > 0 \rightarrow \exists i \in [1 \ldots y]: s = s_{m,i} \wedge (s_{m,i}, e_m, p_{m,i}) \epsilon D_\delta$. Tehát, ha van megadva munkafolyamat szintű korlátozás, valamint ez a δ függvény értelmezési tartományában van, akkor a munkafolyamat sikeresen állapotot válthat.

Metódus szinten tehát, ha nem adunk meg megszorítást, akkor csak a szolgáltatás szintű megszorításokat vesszük figyelembe. Amennyiben metódus szinten valamilyen megszorítás található, akkor azokat is kiértékeljük valamint a szolgáltatás szintű megszorításhalmazzal a metszetképzés művelettel házasítjuk.

4.10 Az esettanulmányok formalizálása

Annak igazolásaképpen, hogy a fenti formalizmust nem öncélúan, l'art pour l'art módon hoztam létre a gyakorlati életet figyelmen kívül hagyva, a korábban bemutatott esettanulmányokat felírom a formalizmus segítségével. Az ily módon felírt követelmények a következő fejezetekben ismertetésre kerülő eszközökkel ellenőrizhetővé válnak.

4.10.1 A postafiók szolgáltatás formalizálása

A 4.5.2 fejezet első példája tartozik ide. Az EDFSM illetve a szerződés definíció leírását már a korábbiakban ismertettem.

Legyen $edfsm_{PF} = (E, S, s_0, P, \delta, F)$ hatos, a következőkben leírtak szerint. Az E eseményhalmaz a következő eseményeket tartalmazza:

- evtQueryPOBox
- evtDownloadDocument
- evtUploadDocument
- evtSendReadReceipt
- evtStop

Az S állapothalmaz a következő elemeket tartalmazza:

- S = StartState (Kezdőállapot)
- PSK = POBoxStateKnown (Postafiók státusz ismert)
- DL = DocumentDownloaded (Dokumentum letöltve)
- RS = ReplySent (Válasz üzenet elküldve)
- RRS = ReadReceiptSent (Olvasási visszaigazolás elküldve)
- E = EndState (Végállapot)

Az *S*-et, mint kezdőállapotot értelmezzük (s0). A *P* halmaz tartalmazza

azokat az előfeltételeket, amelyek az állapotátmenetekhez, események bekövetkezéséhez szükségesek. Esetünkben a $P = \{cnt > 0\}$ egy elemű halmazról beszélünk.

Mindent ismerünk ahhoz, hogy a $\delta: S \times E \times P \rightarrow S$ függvényt meghatározzuk mégpedig úgy, hogy megadjuk azt, hogy a különböző értelmezési tartománybeli diszkrét hármasokhoz milyen értékkészletbeli elem tartozik:

- $\delta(K, \text{evtQueryPOBox}, \emptyset) = PSK$
- $\delta(PSK, \text{evtDownloadDocument}, cnt > 0) = DL$
- $\delta(DL, \text{evtUploadDocument}, \emptyset) = RS$
- $\delta(DL, \text{evtSendReadReceipt}, \emptyset) = RRS$
- $\delta(RS, \text{evtSendReadReceipt}, \emptyset) = RRS$
- $\delta(RRS, \text{evtQueryPOBox}, \emptyset) = PSK$
- $\delta(RRS, \text{evtDownloadDocument}, cnt > 0) = DL$
- $\delta(PSK, \text{evtStop}, cnt = 0) = E$

Az F végállapotok halmaza a következő: {E}.

Legyen C_{PF} a szerződésdefiníció (1. egyenlet), amely meghatározza a metódusok neveit, a szerződés szintű megszorításokat, a metódus szintű megszorításokat, valamint a mögöttes munkafolyamatot:

$$C_{PF} = \left(\left\{ \begin{array}{c} \text{QueryPOBox}, \\ \text{DownloadDocument}, \\ \text{UploadDocument}, \\ \text{SendReadReceipt}, \\ \text{Stop}, \\ \text{GetStatistics}, \\ \text{GetAllocation} \end{array} \right\}, R_{PF}, \left\{ \begin{array}{c} R_{\text{QueryPOBox}}, \\ R_{\text{DownloadDocument}}, \\ R_{\text{UploadDocument}}, \\ R_{\text{SendReadReceipt}}, \\ R_{\text{Stop}}, \\ R_{\text{GetStatistics}}, \\ R_{\text{GetAllocation}}, \end{array} \right\}, \text{edfsm}_{PF} \right)$$

Az R_{PF} fogja jelölni a szolgáltatás szintű megszorításokat (2. egyenlet). Minden metódushoz hozzárendelünk egy megszorítást, illetve megadjuk az edfsm_{PF} állapotgépet amely az állapotátmenet definíciók ismeretével segíti a szolgáltatás működését. Az R_{PF} definícióját tekintve a $T_{\text{ClientAdapter}}$ egy kliens oldali adaptert jelöl, azaz csak ennek a segítségével érhető el a szolgáltatás. Az $\text{Id}_{\text{Registered}}$ és a $\text{net}_{\text{Registered}}$ klasszikus felhasználói és hálózati megszorításokat jelölnek:

$$R_{PF} = (\{T_{\text{ClientAdapter}}\}, \{\text{id}_{\text{Regis tered}}\}, \{\text{net}_{\text{Registered}}\})$$

A postafiók státuszt lekérdező metódus megszorításai a következők:

$$R_{\text{QueryPOBox}} = (\{T_{\text{ClientAdapter}}\}, \{\text{evtQueryPOBox}, \begin{Bmatrix}(S, \emptyset),\\ (\text{RRS}, \emptyset)\end{Bmatrix}\}, \bigwedge \begin{Bmatrix}\text{id}_{\text{Registered}},\\ \text{MU} :: \text{SelfSame}_{\text{QueryPOBox}}\end{Bmatrix}, \{\text{net}_{\text{Registered}}\})$$

A megszorításokat a 4. egyenlet alapján vázoltam. Az $R_{\text{QueryPOBox}}$ metódus szintű megszorítás a $T_{\text{ClientAdapter}}$ kliens oldali adapterből érhető el, azaz nem ad extra megszorításokat a szerződés szintű megszorításokhoz (ez minden további metódusra igaz lesz). A megszorítás $evtQueryPOBox$ eseményt kapcsolja a művelet sikeres végrehajtásához, amely csak abban az esetben aktiválódik, ha a *kezdőállapotban (S)* vagy az *Olvasási visszaigazolás elküldve (RRS)* állapotban van az állapotgép ($\{\text{evtQueryPOBox}, \begin{Bmatrix}(S, \emptyset),\\ (\text{RRS}, \emptyset)\end{Bmatrix}\}$). További követelmény, hogy mindig ugyanazzal a felhasználói azonosítóval hívja meg a kliens a *QueryPOBox* metódust. Ezt az $\wedge \begin{Bmatrix}\text{id}_{\text{Registered}},\\ \text{MU} :: \text{SelfSame}_{\text{QueryPOBox}}\end{Bmatrix}$ kifejezés jelzi. A hálózati megszorításokra vonatkozóan szintén nem rendelkezik további követelményekkel ($\{\text{net}_{\text{Registered}}\}$).

A megszorítás a munkafolyamatra vonatkozó egyszerűsítési szabályokat (4.5.4. fejezet) figyelembe véve:

$$R_{\text{QueryPOBox}} = (\{T_{\text{ClientAdapter}}\}, \{evtQueryPOBox, \{\emptyset\}\}, \bigwedge \begin{Bmatrix}\text{id}_{\text{Registered}},\\ \text{MU} :: \text{SelfSame}_{\text{QueryPOBox}}\end{Bmatrix}, \{\text{net}_{\text{Registered}}\})$$

Mivel nincs megadva előfeltétel ($(\{\text{evtQueryPOBox}, \begin{Bmatrix}(S, \emptyset),\\ (\text{RRS}, \emptyset)\end{Bmatrix}\})$), csak állapot (amelyet az állapotgép egyébként is vizsgál), ezért az $\{evtQueryPOBox, \{\emptyset\}\}$ egyszerűsítés alkalmazható.

A dokumentum letöltésére a következő megszorításokat definiálom:

$$R_{\text{DownloadDocument}} = (\{T_{\text{ClientAdapter}}\}, \{evtDownloadDocument, \begin{Bmatrix}(\text{PSK}, cnt > 0),\\ (\text{RRS}, cnt > 0)\end{Bmatrix}\}, \bigwedge \begin{Bmatrix}\text{id}_{\text{Registered}},\\ \text{MU} :: \text{Same}_{\text{DownloadDocument}}\ (\text{QueryPOBox})\end{Bmatrix}, \{\text{net}_{\text{Registered}}\})$$

A munkafolyamatra vonatkozó egyszerűsítési szabályokat (4.5.4. fejezet) figyelembe véve:

$$R_{\text{DownloadD ocument}} = (\{T_{\text{ClientAdapter}}\}, \{evtDownloadDocument, \{(*, cnt > 0)\}\},$$
$$\bigwedge\left\{\text{MU} :: \text{Same}_{\text{DownloadDocument}}^{\text{id}_{\text{Registered}},}(\text{QueryPOBox})\right\}, \{\text{net}_{\text{Registered}}\})$$

Itt a $evtDownloadDocument$ esemény következik be a sikeres végrehajtás esetében. Előfeltétel az, hogy vagy a *Postafiók státusz ismert (PSK)* vagy pedig az *Olvasási visszaigazolás elküldve* (RRS) állapotban legyen az $edfsm_{PF}$ úgy, hogy legalább egy dokumentum várakozik letöltésre (cnt > 0). A "*" ezeket az állapotokat jelzi a 5. egyenlet alapján. Követelmény még, hogy a dokumentum letöltést ugyanazzal a felhasználó azonosítóval hívja meg a kliens, mint a postafiók lekérdezést ($\wedge\left\{\text{MU} :: \text{Same}_{\text{DownloadDocument}}^{\text{id}_{\text{Registered}},}(\text{QueryPOBox})\right\}$).

A továbbiakban ezen utolsó megállapítás az összes metódusra igaz lesz.

A válasz elküldését végző dokumentum feltöltő műveletnél a $evtUploadDocument$ esemény indukálódik akkor, ha a meghíváskor van letöltött dokumentum. A további feltételek magyarázatát már az előzőekben megadtam, az egyszerűsített definíciót közlöm:

$$R_{\text{UploadDocument}} = (\{T_{\text{ClientAdapter}}\}, \{evtUploadDocument, \{\emptyset\}\},$$
$$\bigwedge\left\{\text{MU} :: \text{Same}_{\text{UploadDocument}}^{\text{id}_{\text{Registered}},}(\text{QueryPOBox})\right\}, \{\text{net}_{\text{Registered}}\})$$

Az olvasási visszaigazolás akkor küldhető, ha előzőleg sikeres dokumentum letöltés történt, vagy pedig ezután még egy válasz üzenetet is elküldtünk (itt találkozik az állapotgép ábráján is jól látható két lehetséges út):

$$R_{\text{SendReadReceipt}} = (\{T_{\text{ClientAdapter}}\}, \{evtSendReadReceipt, \{\emptyset\}\},$$
$$\bigwedge\left\{\text{MU} :: \text{Same}_{\text{SendReadReceipt}}^{\text{id}_{\text{Registered}},}(\text{QueryPOBox})\right\}, \{\text{net}_{\text{Registered}}\})$$

A folyamat akkor áll meg, ha lekérdeztük a postafiók állapotát és az nem jelzett letölthető dokumentumot (cnt > 0):

$$R_{\text{Stop}} = (\{T_{\text{ClientAdapter}}\}, \{evtStop, \{(*, cnt = 0)\}\},$$
$$\bigwedge\left\{\text{MU} :: \text{Same}_{\text{Stop}}^{\text{id}_{\text{Registered}},}(\text{QueryPOBox})\right\}, \{\text{net}_{\text{Registered}}\})$$

Természetesen olyan metódusok is szerepelhetnek a szerződésben, amelyek nem kapcsolódnak a konkrét munkafolyamathoz.

Esetünkben ezek a *GetStatistics*, amely a postafiók használat statisztikáját adja vissza, valamint a *GetAllocation*, amely a postafiók foglaltsági adatokat jelzi vissza.

A *GetStatistics* metódus megszorításai a következők:

$$R_{GetStatistics} = (\{T_{ClientAdapter}\}, \{\}, \{id_{Registered}\}, \{net_{Registered}\})$$

Az GetAllocation metódusé hasonlóak:

$$R_{GetAllcation} = (\{T_{ClientAdapter}\}, \{\}, \{id_{Registered}\}, \{net_{Registered}\})$$

A letölthető dokumentumok száma nyilvánvalóan minden letöltéskor eggyel csökken.

4.10.2 Az elfogadási munkafolyamat formalizálása

Legyen $edfsm_{ACC} = (E, S, s_0, P, \delta, F)$ hatos, a következőkben leírtak szerint. Az E eseményhalmaz a következő eseményeket tartalmazza:

- evtStartWork
- evtSendToAcceptance
- evtAccept
- evtReject

Az S állapothalmaz a következő elemeket tartalmazza:

- S = StartState (Kezdőállapot)
- SW =StartedToWork (Munkavégzés folyamatban)
- STA = SentToAcceptance (Elfogadás alatt)
- E = EndState (Végállapot)

Az s0, mint kezdőállapot a *S*, a P halmaz üres.

A δ: S × E × P → S függvényt a következőképpen definiálom:

- $\delta(K, evtStartWork, \emptyset) = SW$
- $\delta(SW, evtSendToAcceptance, \emptyset) = STA$
- $\delta(STA, evtAccept, \emptyset) = E$
- $\delta(STA, evtReject, \emptyset) = SW$

Az F végállapotok halmaza a következő: {E}.

Legyen C_{Acc} a szerződésdefiníció (1. egyenlet):

$$C_{Acc} = \left(\left\{ \begin{array}{c} \text{StartWork}, \\ \text{SendToAcceptance}, \\ \text{Accept}, \\ \text{Reject}, \\ \text{GetHistory}, \\ \text{GetRunningWorkFlows} \end{array} \right\}, R_{Acc}, \left\{ \begin{array}{c} R_{\text{StartWork}}, \\ R_{\text{SendToAcceptance}}, \\ R_{\text{Accept}}, \\ R_{\text{Reject}}, \\ R_{\text{GetHistory}}, \\ R_{\text{GetRunningWFs}}, \end{array} \right\}, edfsm_{Acc} \right)$$

A szolgáltatás szintű megszorításokat (2. egyenlet) most is az alapértelmezett tulajdonságokkal látom el:

$$R_{Acc} = (\{T_{\text{ClientAd apter}}\}, \{id_{\text{Registered}}\}, \{net_{\text{Registered}}\})$$

A munkavégzés kezdetét jelző metódus az $evtStartWork$ eseményt indukálja, a kezdőállapotból hajtható végre. Ezt a műveletet csak beosztottak végezhetik, akik az $id_{Employees}$ csoport tagjai:

$$R_{\text{StartWork}} = (\{T_{\text{ClientAdapter}}\}, \{evtStartWork, \{\emptyset\}\}, \{id_{\text{Employees}}\}, \{net_{\text{Registered}}\})$$

Az alkalmazott a főnök részére elfogadásra küldi az elvégzett munkát, vagyis az $evtSendToAcceptance$ eseményt indukálja a művelet akkor, ha a munkavégzés folyamatban van. További elvárás az, hogy a műveletet csak beosztottak végezhessék el, valamint követelmény, hogy ugyanaz a beosztott végezze a munka elfogadásra küldését, mint aki a munkát ténylegesen elvégezte ($\wedge \left\{ \begin{array}{c} id_{\text{Employees}}, \\ MU :: Same_{\text{SendToAcceptance}} \text{ (StartWork)} \end{array} \right\}$):

$$R_{\text{SendToAcceptance}} = (\{T_{\text{ClientAdapter}}\}, \{evtSendToAcceptance, \{\emptyset\}\}, \bigwedge \left\{ \begin{array}{c} id_{\text{Employees}}, \\ MU :: Same_{\text{SendToAcceptance}} \text{ (StartWork)} \end{array} \right\}, \{net_{\text{Registered}}\})$$

Az elfogadás művelet az *Elfogadás Alatt (STA)* állapotban hajtható végre. Megköveteljük, hogy a *Bosses* csoport tagjai végezhessék a műveletet, valamint a műveletet végző természetes személy valóban a felettese (*Sup*) legyen annak, aki a munkát végezte (*StartWork*). Amennyiben előzőleg már elutasításra került sor, akkor megköveteljük, hogy a beosztott ugyanazon főnöke (ha több van) végezhesse csak az elfogadást, aki a munkát visszautasította ($\wedge \left\{ \begin{array}{c} id_{\text{Bosses}}, \\ MU :: Sup_{\text{Accept}} \text{ (StartWork)}, \\ MU :: Same_{\text{Accept}} \text{ (Reject)} \end{array} \right\}$):

$$R_{\text{Accept}} = (\{T_{\text{ClientAdapter}}\}, \{evtAccept, \{\emptyset\}\}, \bigwedge \begin{Bmatrix} id_{\text{Bosses}}, \\ MU :: Sup_{\text{Accept}}(\text{StartWork}), \\ MU :: Same_{\text{Accept}}(\text{Reject}) \end{Bmatrix}, \{net_{\text{Registered}}\})$$

A visszautasítás műveletre hasonló megállapítások tehetők, mint az elfogadásra:

$$R_{\text{Reject}} = (\{T_{\text{ClientAdapter}}\}, \{evtReject, \{\emptyset\}\}, \bigwedge \begin{Bmatrix} id_{\text{Bosses}}, \\ MU :: Sup_{\text{Reject}}(\text{StartWork}), \\ MU :: SelfSame_{\text{Reject}} \end{Bmatrix}, \{net_{\text{Registered}}\})$$

A *GetHistory* illetve a *GetRunningWorkFlows* munkafolyamat és állapot független metódusok:

A *GetHistory* metódus a munkafolyamat eddigi történéseit adja vissza. Ezt a metódust csak a menedzserek futtathatják:

$$R_{\text{GetHistory}} = (\{T_{\text{ClientAdapter}}\}, \{\}, \{id_{\text{Managers}}\}, \{net_{\text{Registered}}\})$$

Az *GetRunningWorkflows* segítségével a futó munkafolyamatokról szerezhetünk információt:

$$R_{\text{GetRunningWFs}} = (\{T_{\text{ClientAdapter}}\}, \{\}, \{id_{\text{Managers}}\}, \{net_{\text{Registered}}\})$$

4.11 Az elért eredmények összegzése

Az 3. fejezetben a modern elosztott alkalmazások architekturális és folyamat-módszertani elemzésére tértem ki. Fontosnak tartom a Szolgáltatás Orientált Architektúra, a többrétegű tervezés alapelveit a szoftverarchitektúra vonatkozásában, amelyek az objektum orientált és a komponensorientált fejlesztési paradigmák és az ide kapcsolódó tervezési minták extrapolációi. Ezek után az agilis illetve iteráció alapú, a konkrét fejlesztendő terméket a fókuszba állító folyamatmodellekről és ezek építőelemeiről beszéltem. Ennek a néhány oldalas bevezető írásnak a szerepe az, hogy behelyezzem a valódi környezetébe a 4. fejezetekben leírtakat.

A 4. fejezetben egy olyan általam kidolgozott módszert [8] mutatok be, amely más kutatók által készített munkák továbbfejlesztésének fogható fel, illetve a már meglévő, de eddig egymástól függetlenül kezelt biztonsági megszorítások

felhasználását rendszerezi.

Bevált gyakorlat szerint a homlokzat (facade) tervezési minta [37] segítségével egy interfészt adunk a külvilág számára, amely egy rendszer szolgáltatásait képes publikálni a külvilág felé.

Ezek a publikált szolgáltatások semmilyen megszorításokat nem definiálnak a szerződés szintjén, amelyet megoldandó problémának látok. Ezért kidolgoztam egy módszert, amelynek segítségével megszorításokkal láthatók el a kliens és a szerver kommunikációja során definiált szerződések

A módszert az üzleti szolgáltatások és a mögöttük üzemelő munkafolyamatok, a hívó felhasználóra vonatkozó dinamikus szabályok, a futás idejű hozzáférés-vezérlés kiterjesztése, hálózati korlátozások ötvözése jellemzi.

Ebből kifolyólag egy olyan formálisan is leírható, illetve definiálható üzleti szolgáltatásokra vonatkozó megszorításkészletet illetve ezeket ellenőrző mechanizmusokat alakítottam ki, amelyek a gyakorlatban felmerült problémákra adnak megoldást.

A fejezetben két tipikus példát, esettanulmányt említek meg, majd körültekintő leírás után mindkettőt formálisan is definiálom. Mindkettőhöz formálisan megadom a megszorításaikat is, ezzel bizonyítva a formális rendszerem életképességét.

A következő (5.) fejezetben implementálom a bemutatott validációs keretrendszert, valamint az esettanulmányokat. A formalizmus kialakítása során azt is szem előtt tartottam, hogy belőle egy ipari környezetben is hasznosítható szoftvereszköz váljon.

1. Tézis. Megmutattam a jelenlegi elosztott alkalmazások esetében használatos hozzáférés-vezérlő mechanizmusokban található korlátokat. Egy olyan formális modellt definiáltam, amely megválaszolja az elosztott hozzáférés-vezérlés legfontosabb kérdéseit, összekapcsolja a szolgáltatások és munkafolyamatok jogosultságkezelését, kiterjeszti a szerepkör alapú valamint a hálózati szegmensekhez tartozó jogosultságkezelést, valamint absztraktságából adódóan garantálja a platform- és implementációfüggetlenséget. A formális modell alkalmazhatóságát több ipari esettanulmányon keresztül is validáltam.

A témával kapcsolatos eredményeket [4] [7][8] alatt publikáltam.

5 A formális modell megvalósítása

Nevezhetném ezt a fejezetet „megvalósíthatósági tanulmánynak” is, azonban ezt a címet olyan dokumentumoknak szokás adni, amelyek egy előzőleg eltervezett (nem feltétlenül informatikai) projektről nyújtanak információt olyan döntéshozók számára, akik legtöbbször a projekt finanszírozásáról döntenek. A megvalósíthatósági tanulmány feladata a projekt illetve módszer megalapozottságának, végiggondoltságának vizsgálata. A megvalósíthatósági tanulmányban sokszor egy vagy több alternatív megoldás is számba vehető, amelyeket erőforrásbecsléssel, specifikáció alapú szempontokkal támogatva összehasonlíthatóvá válnak.

Jelen esetben szó sincs finanszírozási kérdésekről, vagy éppen alternatív megoldások kereséséről. Ebben a fejezetben a 4. fejezetben bevezetett formalizmus alapján egy Microsoft .NET keretrendszerre épülő megoldás legmodernebb, legszakszerűbb informatikai eszközökkel való megvalósíthatóságát vizsgálom, és megmutatom azt a végkifejletet, amely szigorúan végiggondolt architekturális és implementációs döntések során alakult ki.

5.1 Célkitűzések, irányok

Célom a 4. fejezetben bevezetett jelölésrendszer alapján egy olyan a gyakorlatban is alkalmazható keretrendszer kidolgozása [8], amely a biztonsági rendszer formális definíciója után egyszerű átírási lépések sorozatával működő rendszerré tud összeállni.

Nem mindegy, hogy a végeredmény mennyire átlátható, mennyire könnyen módosítható, mekkora kifejező erővel rendelkezik. Ezért a 3.4. fejezetben számba vettem a legfontosabb programozási paradigmákat, amelyek felhasználhatóak lehetnek a keretrendszer tervezésekor, illetve implementációjakor. Az objektumorientált paradigma a keretrendszer működési modelljét írja le, meghatározza, hogy melyik osztály milyen feladattal, szerepkörrel rendelkezik, ezek hogyan kapcsolódnak egymáshoz. Amikor attribútumokat fogok felhasználni, akkor már deklaratív programozási elemeket alkalmazok. Valamiféle, aspektus-orientált programozás-szerű megoldás is szükségessé válik a keretrendszer implementációjakor. Amire az adott szcenárióban igény van az aspektus-orientált szemléletmódból, azt a .NET keretrendszer beépített eszközeivel is megoldható.

Ezek után a C# nyelv attribútum fogalmának [84][52] felhasználásával a formális megkötéseket C# attribútumoknak feleltetem meg. Itt a deklaratív

programozás elveit használom ki. Az attribútumok létrehozását az attribútum nyelvtanok elmélete [26] ihlette.

A szerződés definícióját C# nyelvi interfészek segítségével adom meg. Ez a legelterjedtebb módja újrafelhasználható komponensek definiálásának, amelyet az objektumorientált és a komponens alapú paradigma elvei szerint végzek. A dolgozatban bemutatott két gyakorlati példát átírom a fentiek szerint, valamint a .NET 3.0-ban bevezetett WF (Workflow Foundation) [109] segítségével definiálom a kapcsolódó munkafolyamatokat, állapotgépeket.

Mivel elosztott rendszerről beszélünk, ezért a metódushívásoknak folyamat, gép vagy akár hálózat határokon kell átívelniük. Ismertetem a .NET 3.0-ban bevezetett WCF (Windows Communication Foundation) [108] kommunikációs „paradigma" elveit, majd pedig megmutatom, hogy a WCF részét képező funkcionalitások miként használhatók fel a keretrendszer implementációjakor alkalmazható aspektus orientált elveket tükröző szolgáltatásokhoz.

Ezáltal kialakul a végleges architektúra, amelyet egy ábra segítésével is szemléltetek 5.5-ban.

5.2 Attribútumok

Vegyük sorra azokat az absztrakt definíciókat, amelyeket a következő fejezetekben bevezettem: 0, 4.6, 4.7, 4.8. Azokat a formális építőelemeket, amelyek a szerződéshez illetve a szerződés építőelemeihez deklaratív információt adnak hozzá, C# attribútumok segítségével fogom jelölni.

A szerződés definíciója (C) során feltüntetett $m_1, m_2, \ldots, m_n$ metódusok a szerződés építőelemeinek, vázának tekinthetők, ezekhez illetve magához a szerződéshez kell attribútumokat kapcsolni. A szerződés szintű R_C, illetve a különböző metódusokhoz tartozó $R_{m_1}, R_{m_2}, \ldots, R_{m_n}$ megszorításokban található elemeket veszem sorba.

A gyakorlatban a fejlesztést egy fejlesztői környezetben végezzük, amely sok aspektusában eltérhet attól az éles rendszertől, ahol majd a kész szoftver üzemelni fog. Ezt a különbözőséget úgy szoktuk feloldani, hogy a változó paramétereket konfigurációs állományokban rögzítjük. Amennyiben a továbbiakban ismertetésre kerülő attribútum rendelkezik *FromConfig* logikai tulajdonsággal, akkor az a következő módon módosítja az attribútum kiértékelésének menetét:

1. Ha a tulajdonság értéke *hamis*, akkor az egy darab szöveges paramétert váró konstruktorban megadott érték kerül kiértékelése.
2. Ha a tulajdonság értéke *igaz*, akkor a szöveges érték által jelzett konfiguráció kulcs érték párja kerül kiértékelésre.

Az attribútumok ismertetésénél mindig jelzem majd, hogy hol kap szerepet a *FromConfig* tulajdonság.

5.2.1 A munkafolyamat és lépéseinek szerződéshez kötése

De még mielőtt mindezekre rátérnék a szerződés definíciójában található *edfsm* munkafolyamatnak és magának a szerződésnek az összekapcsolását kell megoldani. A következő attribútumot vezetem be:

```
[AttributeUsage(AttributeTargets.Class)]
public class StateMachineAttribute : System.Attribute
{
    public Type WorkFlowController { get; private set; }

    public StateMachineAttribute(Type workFlowController)
    {
        WorkFlowController = workFlowController;
    }
}
```

10. ábra: Állapotgép attribútum

Az fenti kódrészletből leolvasható, hogy minden attribútumnak a *System.Attribute* osztályból kell származnia. Paramétereket az osztályoknál megszokott módon többek között a konstruktorán keresztül kaphat. A .NET konvenció az, hogy az attribútumot definiáló osztály nevének az *Attribute* szóra kell végződnie, amely felhasználáskor opcionálisan el is hagyható. Azt, hogy miért csak osztályokhoz kapcsolható a *StateMachineAttribute*, és miért nem interfészekhez, az 5.3. fejezetben ismertettem.

Az *IWorkFlowController* interfészt a következőképpen hoztam létre, amely egyedül az állapotgép munkafolyamat típusát tartalmazza:

```
public interface IWorkFlowController
{
    Type StateMachineWorkFlow { get; }
}
```

11. ábra: Munkafolyamat vezérlő interfész

A típus alapján dinamikusan kerül létrehozásra az állapotgép. Az *IWorkFlowController* interfész későbbi konkrét implementációja fog minden olyan információt tartalmazni fog, amely a munkafolyamathoz szükséges.

Az üzleti interfész metódusai elé a *StateMachineStepAttribute* attribútumot helyezhetjük el. Az attribútum a konstruktorában megadja, hogy milyen eseményt fog a metódushívás kiváltani a szolgáltatás mögött futó munkafolyamatban:

```
[AttributeUsage(AttributeTargets.Method)]
public class StateMachineStepAttribute : System.Attribute
{
    public string RaisedEvent { get; private set; }

    public StateMachineStepAttribute(string raisedEvent)
    {
        this.RaisedEvent = raisedEvent;
    }
}
```

12. ábra: Munkafolyamat esemény attribútum

Egy vagy több *StateAndPreConditionAttribute* attribútum meghatározza azokat az (legalább egy darab) állapot-előfeltétel párokat, amelyek valamelyikének teljesülése esetében a munkafolyamat állapotot válthat:

```
[AttributeUsage (AttributeTargets.Method, AllowMultiple=true)]
public class StateAndPreConditionAttribute : System.Attribute
{
    // ...

    public StateAndPreConditionAttribute(string state)
    {
        this.State = state;
    }

    public StateAndPreConditionAttribute(string state,string pcm)
    {
        this.State = state;
        this.PreConditionMehod = pcm;
    }
}
```

13. ábra: Munkafolyamat állapot és előfeltétel attribútum

A *state* paraméter a formalizmusnak megfelelően a * értéket is felveheti.

Mivel az attribútumok paraméterei csak egyszerű típusok, illetve ezekből álló tömbök lehetnek [14], ezért egy dedikált attribútumot (*StateAndPreConditionAttribute*) vezettem be állapot-előfeltétel pároknak. Ha ez a nyelvi limitáció nem korlátozna, akkor ezeket az információkat is a *StateMachineStepAttribute* attribútumon keresztül adnám meg

Az előfeltételt egy logikai értéket visszaadó metódus szöveges formátumban jelölt nevének formájában adom meg (*PreConditionMethod*). Ennek okai az alábbiakban láthatók:

1. Egyszerű C# nyelvű kódrészlet megadása a gyakorlatban túlzott kötöttségeket jelentene, ezért egy metódus megadása szükséges.
2. Mivel a *StateAndPreCondition* osztály egy attribútum paramétereként fog megjelenni, és a C# nyelv nem támogatja azt, hogy attribútum

paraméterébe metódusreferenciát helyezzünk el, ezért egy metódus nevet adhatunk meg, amelynek az aktuális implementáció szerint szerepelnie kell a *C* interfészt implementáló osztályban.

Fontos még megjegyezni, hogy amennyiben az attribútummal annotált metódus nem fut le sikeresen (azaz kivételt dob), akkor az aktuális implementációban az állapotátmenet sem hajtódik végre.

5.2.2 A hívó típusának, felhasználói azonosítójának, hálózati helyének attribútumai

Áttérek a szolgáltatás és a metódus szintű megszorításokban felhasznált hívó típusára vonatkozó, felhasználói azonosító illetve hálózati szintű megszorításokra.

A hívó típusára az *AllowedCallerType* attribútumot vezetem be (több is megadható egy interfész illetve metódus előtt):

```
[AttributeUsage(AttributeTargets.Interface |
                AttributeTargets.Method, AllowMultiple = true)]
public class AllowedCallerTypeAttribute : System.Attribute
{
    // ...

    public bool FromConfig { get; set; }

    public AllowedCallerTypeAttribute(Type caller)
    {
        this.CallerType = caller;
    }

    public AllowedCallerTypeAttribute(string caller)
    {
        this.CallerTypeName = caller;
    }
}
```

14. ábra: Engedélyezett hívó típus attribútum

Az attribútumot interfészre (a szerződésre) illetve metódusokra (a szerződés elemei) lehet ráakasztani az *AttributeUsage* attribútum beállításai szerint.

Kétfajta konstruktort is definiáltam, az egyik esetében ismerjük, és ezáltal fel tudjuk oldani már fordítási időben a típust (*typeof* kulcsszó), míg a másik esetben nem. Az első konstruktor jól használható a szerver készítője által létrehozott, a kliens részére kibocsátott adapter (4.6. fejezet) esetében, ugyanis ekkor a szerver és a kliens is ismeri az adapter implementációjakor definiált típust, ahonnan a szerver hívása kizárólagosan megtörténhet.

Amennyiben a *FromConfig* logikai változó értéke igaz, akkor a fenti példában szereplő második konstruktor *string* típusú *caller* paramétere nem a konkrét típust tartalmazza, hanem egy hivatkozást arra a konfigurációs

bejegyzésre, amely a futás idejű típus értékét tartalmazza.

A következő témakör a hívó felhasználói identitásának korlátozása. A .NET Framework esetében használható a *PrincipalPermissionAttribute* [51] nevű attribútum is, amely szorosan integrálható a WCF-es szolgáltatásokkal. Amennyiben ezt az attribútumot kívánjuk használni, lehetőségünk van rá. A következőkben definiálásra kerülő *AllowedIdentityAttribute* illetve az *AllowedRoleAttribute* úgy váltja ki a beépített attribútumot, hogy az magasabb fokú integrációt biztosítson az általam létrehozott megoldással.

Minden felhasználói jogosultsághoz tartozó attribútum az *AllowedIdentiyBaseAttribute* attribútumból származik. A *FromConfig* ebben az esetben is hasonlóan működik, mint az *AllowedCallerTypeAttribute* attribútumnál:

```
public abstract class AllowedIdentityBaseAttribute :
                                                    System.Attribute
{
    public bool FromConfig { get; set; }
}
```

15. ábra: Engedélyezett felhasználói azonosító alapattribútum

Az *AllowedIdentityAttribute* illetve az *AllowedRoleAttribute* attribútumok a felhasználói illetve csoport szintű engedélyeket vezérlik. Mindkét attribútumból természetesen több is megadható:

```
[AttributeUsage(AttributeTargets.Interface |
                AttributeTargets.Method, AllowMultiple = true)]
public class AllowedIdentityAttribute :
                                        AllowedIdentityBaseAttribute
{
    public string[] Identities { get; private set; }

    public AllowedIdentityAttribute(params string[] identities)
    {
        this.Identities = identities;
    }
}

[AttributeUsage(AttributeTargets.Interface |
                AttributeTargets.Method, AllowMultiple = true)]
public class AllowedRoleAttribute : AllowedIdentityBaseAttribute
{
    public string[] Roles { get; private set; }

    public AllowedRoleAttribute(params string[] roles)
    {
        this.Roles = roles;
    }
}
```

16. ábra: Engedélyezett felhasználó attribútumok

A formális definíció során már megállapítottam, hogy ugyanazon, illetve különböző metódusok egymás után történő meghívását végző felhasználók közötti korreláció kizárólag metódus szinten határozható meg.

Az ide kapcsolódó formális szabályokat a következő két C# felsorolási típusban rögzítettem:

```
public enum SelfRule
{
    SelfSame = 1,
    SelfDiff = 2,
    SelfSubst = 3,
}

public enum RelationRule
{
    Same = 1,
    Diff = 2,
    Subst = 3,
    Sup = 4,
    Inf = 5
}
```

17. ábra: Szabály enumerációk

Azon attribútum, amely a szabályokat fogadja (*AllowedRuleAttribute*) ennek megfelelően két konstruktorral rendelkezik. Az egyik konstruktor egy *SelfRule* enumerációt vár, ugyanis a szabályban résztvevő metódus mindig az, amelyet annotálunk az attribútum segítségével. A másik konstruktor pedig egy *RelationRule* enumerációt, illetve annak a metódusnak a nevét fogadja, amely a másik résztvevő a kétparaméteres szabály kiértékelése során. Fontos megjegyezni, hogy ennek az attribútumnak a használatakor nem hatásos a *FromConfig* tulajdonság, ugyanis itt minden paraméter a munkafolyamat végrehajtási környezetéből elérhető:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class AllowedRuleAttribute : AllowedIdentityBaseAttribute
{
    // ...

    public AllowedRuleAttribute(SelfRule rule)
    {
        this.SR = rule;
    }

    public AllowedRuleAttribute(RelationRule rule,
                                                string otherMethod)
    {
        this.RR = rule;
        this.OtherMethod = otherMethod;
    }
}
```

18. ábra: Engedélyezett felhasználó-szabály attribútum

A figyelmes olvasó észrevehette, hogy az előbb felsorolt három attribútum között nem definiáltam logikai összefüggéseket (minden esetben a logikai *és* műveletet használjuk), nem kapcsoltam össze az attribútum által hordozott információt logika műveletekkel annak ellenére, hogy a formalizmusban erre lehetőséget adtam. A keretrendszerben erre, mint továbbfejlesztési lehetőségre tekintek.

A hálózati megszorításokat biztosító *AllowedNetworkAttribute* attribútum használható interfész és metódus szinten. Háromfajta konstruktorral rendelkezik:

1. IP címet vár
2. IP címet és hálózati maszkot vár
3. Szöveges paraméter értéket vár, amely
    a. Ha hamis a FromConfig értéke, akkor a számítógép nevét,
    b. Ha igaz, akkor pedig annak a konfigurációs bejegyzésnek a nevét, ahonnan a hálózati beállításokat kiolvassa a keretrendszer tartalmazza.

Ez az attribútum az előzőekhez képest nem rendelkezik újdonságtartalommal, ezért a forráskódját nem emelem be, csak a fenti definíciót adom a működésére.

5.3 Az esettanulmányok munkafolyamat- és szerződésdefiníciója

Az előzőekben ismertettem a szerződésdefiníció során használható attribútumokat. Adós vagyok a munkafolyamatok modellezőeszközének ismertetésével, továbbá a példák munkafolyamatainak illetve szerződéseinek leírásával.

Amikor a konkrét esetek bemutatása során a szükséges alap építőelemeket vázolom, akkor nem fogom pontosan elhelyezni őket a komponens szinten. Erre, mivel egy kiemelt témakörről beszélünk, egy külön fejezetet szánok. Ahol fontosnak látom, ott annyit fogok megemlíteni, hogy szerver oldali vagy pedig kliens oldali építőelemről beszélünk-e.

Mint ahogy a korábbiakban utaltam rá, az üzleti szolgáltatások mögött futó munkafolyamatokat Workflow Foundation (WF) [109] segítségével fogom definiálni. Mivel egy olyan technológiáról beszélünk, ami tudományos szempontból érdektelen, ezért csak röviden szólok róla. A WF állapotgép illetve szekvenciális munkafolyamatok definiálását teszi lehetővé. Az állapotgépek esetében megvalósítja az állapot illetve az esemény absztrakt fogalmát. Támogatja a hosszan tartó munkafolyamatokat beépített, sőt tetszőlegesen kiegészíthető perzisztencia szolgáltatások segítségével.

Maga a WF nem támogatja a felhasználók, csoportok, jogosultságok kezelését, a feladatkiosztást. Ezek már magasabb üzleti szinten értelmezhető illetve értelmezett fogalmak. A WF használata mellett a hosszú munkafolyamatok kezelése illetve a beépített perzisztencia-szolgáltatás megléte tette le a voksot. Ezeket a szolgáltatásokat csak sokórányi fejlesztés során érhettem volna el egy State (állapot) tervezési mintát [37] vagy éppen egy felsőháromszög-mátrixban definiált munkafolyamat esetében.

5.3.1 Postafiók szolgáltatás

A 4.10.1 fejezetben formalizáltam a postafiók szolgáltatást a korábban felvázolt formalizmus segítségével. A formalizált szerződésdefiníció egyértelműen átírható attribútumokkal ellátott C# interfész formájába:

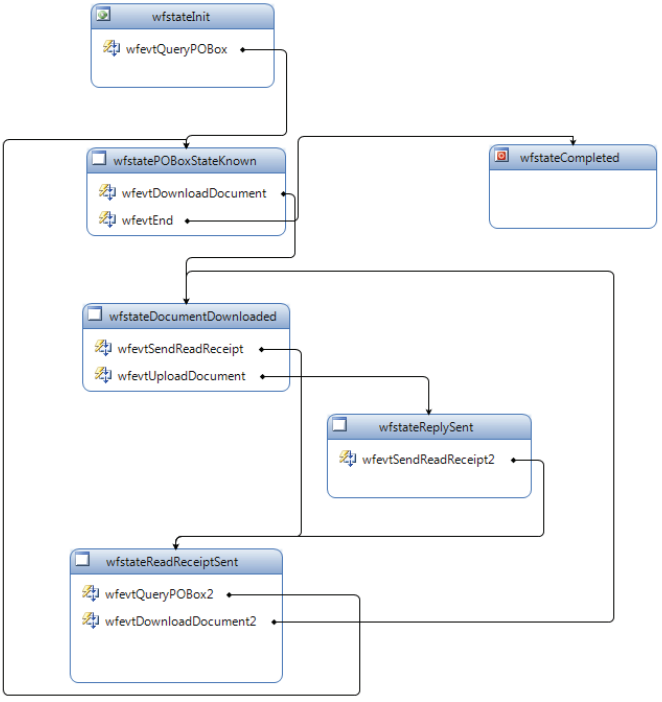
```
[AllowedCallerType("HKPClientLib", FromConfig = true)]
[AllowedIdentity("HKPIdentities", FromConfig = true)]
[AllowedNetwork("HKPNetwork", FromConfig = true)]
[ServiceContract]
public interface IHKPService
{
    [StateMachineStep("evtQueryPOBox")]
    [AllowedIdentity("HKPIdentities", FromConfig = true)]
    [AllowedRule(SelfRule.SelfSame)]
    [OperationContract]
    POBoxStateInfo QueryPOBox(POBox poBox);

    [StateMachineStep("evtDownloadDocument")]
    [StateAndPreCondition("*", "HasNewDocuments")]
    [AllowedIdentity("HKPIdentities", FromConfig = true)]
    [AllowedRule(RelationRule.Same, "QueryPOBox")]
    [OperationContract]
    Document DownloadDocument(POBox poBox);

    [StateMachineStep("evtUploadDocument")]
    [AllowedIdentity("HKPIdentities", FromConfig = true)]
    [AllowedRule(RelationRule.Same, "QueryPOBox")]
    [OperationContract]
    void UploadDocument(POBox poBox, Document reply);

    [StateMachineStep("evtSendReadReceipt")]
    [AllowedIdentity("HKPIdentities", FromConfig = true)]
    [AllowedRule(RelationRule.Same, "QueryPOBox")]
    [OperationContract]
    void SendReadReceipt(POBox poBox, int documentId);

    [StateMachineStep("evtStop")]
    [StateAndPreCondition("*", "NoNewDocuments")]
    [AllowedIdentity("HKPIdentities", FromConfig = true)]
    [AllowedRule(RelationRule.Same, "QueryPOBox")]
    [OperationContract]
    void Stop(POBox poBox);

    [OperationContract]
    POBoxStatistics GetStatistics(POBox poBox, TimeFrame tFrame);

    [OperationContract]
    POBoxAllocation GetAllocation(POBox poBox);
}
```

19. ábra: HKP szerződésdefiníció

A *ServiceContract* és az *OperationContract* attribútumokat eddig még nem ismertettem. Ezek a WCF kommunikációs platform miatt lényegesek. Az attribútumok jelzik a szolgáltatási szerződést illetve a szerződés keretein belül publikált metódusokat. A *StateMachineStep* illetve a *StateAndPreCondition* attribútumok paraméterértékeinek forrását a következő bekezdésekben ismertetem.

Mint ahogy korábban említettem, nem biztos, hogy minden üzleti metódus a munkafolyamat részeként értelmezett. Ilyen a fenti példában a *GetStatistics* illetve a *GetAllocation* metódus is, ugyanis a postafiók statisztikát, illetve foglaltságot tetszőleges időpontban és állapotban lekérdezhetjük.

A WF munkafolyamatokat Visual Studio-ba épített modellező eszköz segítségével lehet elkészíteni:



20. ábra: HKP munkafolyamat WF-ben

A *StateAndPreCondition* attribútum paraméterei a fenti munkafolyamat állapotaiból, illetve a szerződés implementációjában később látható logikai értéket visszaadó függvényekből származnak.

A munkafolyamat definíciója önmagában még nem elég. Egy *IWorkflowController* interfészből származó *IHKPController* interfészt párosítok hozzá, amelyet a *HKPController* vezérlő osztály implementál:

```
public class HKPController : IHKPController
{
    #region IHKPController Members
    public event EventHandler<ExternalDataEventArgs>
                                                evtQueryPOBox;
    public event EventHandler<ExternalDataEventArgs>
                                                evtDownloadDocument;
    public event EventHandler<ExternalDataEventArgs>
                                                evtUploadDocument;
    public event EventHandler<ExternalDataEventArgs>
                                                evtSendReadReceipt;
    public event EventHandler<ExternalDataEventArgs> evtStop;
    #endregion

    #region IWorkFlowController Members
    public Type StateMachineWorkFlow
    {
        get { return typeof(HKPWorkFlow); }
    }
    #endregion
}
```

21. ábra: HKP munkafolyamat vezérlő

Az *IHKPController* interfész eseményei kapcsolódnak össze a munkafolyamat állapotátmeneteivel illetve a szerződésdefinícióban megadott *StateMachineStep* attribútumok paraméterivel. A *StateMachineWorkFlow* tulajdonság hordozza annak a munkafolyamatnak a típusát, amely a háttérben levezényli a folyamatokat.

Magát az üzleti szolgáltatást, a *HKPService* osztály implementálja:

```
[StateMachine(typeof(HKPController))]
public class HKPService : IHKPService
{
    //
    // IHKPService Members
    //

    public bool HasNewDocuments()
    {
        // ...
    }

    public bool NoNewDocuments()
    {
        return !HasNewDocuments();
    }
}
```

22. ábra: HKP szolgáltatás

Mivel az állapotgépet szerver oldalon definiálom, ezért a szerver oldalon elérhető szolgáltatásimplementáció attribútuma segítségével kapcsolódik össze a munkafolyamat vezérlő és az üzleti szolgáltatás maga.

5.3.2 Elfogadási munkafolyamat

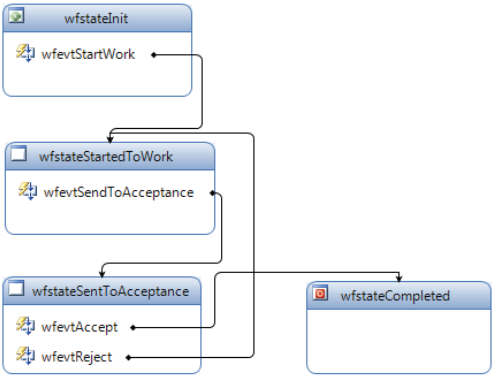
Az elfogadási munkafolyamat formalizmusának részletezésére a 4.10.2. fejezetben került sor. Mivel a Postafiók szolgáltatásnál már minden fontos tudnivalót ismertettem a rendszer fizikai komponenseinek definíciója kapcsán, most csak felsorolásszerűen mutatom be ezeket.

A C# nyelvű interfészként ábrázolt szerződésdefiníciót a következő kódrészlet mutatja:

```
[AllowedCallerType("AccClientLib", FromConfig = true)]
[AllowedIdentity("AccIdentities", FromConfig = true)]
[AllowedNetwork("AccNetwork", FromConfig = true)]
[ServiceContract]
public interface IAccService
{
    [StateMachineStep("evtStartWork")]
    [AllowedIdentity("AccIdentities", FromConfig = true)]
    [OperationContract]
    WorkPiece StartWork();

    [StateMachineStep("evtSendToAcceptance")]
    [AllowedIdentity("AccIdentities", FromConfig = true)]
    [AllowedRule(RelationRule.Same, "StartWork")]
    [OperationContract]
    void SendToAcceptance(WorkPiece wp);

    [StateMachineStep("evtAccept")]
    [AllowedIdentity("AccBosses", FromConfig = true)]
    [AllowedRule(RelationRule.Same, "Reject")]
    [AllowedRule(RelationRule.Sup, "StartWork")]
    [OperationContract]
    void Accept(WorkPiece wp);

    [StateMachineStep("evtReject")]
    [AllowedIdentity("AccBosses", FromConfig = true)]
    [AllowedRule(RelationRule.Sup, "StartWork")]
    [AllowedRule(SelfRule.SelfSame)]
    [OperationContract]
    void Reject(WorkPiece wp);

    [OperationContract]
    [AllowedIdentity("AccManagers", FromConfig = true)]
    History GetHistory(WorkPiece wp);

    [OperationContract]
    [AllowedIdentity("AccManagers", FromConfig = true)]
    WorkflowContainer GetRunningWorkflows();
}
```

23. ábra: Elfogadási szolgáltatás szerződése

A munkafolyamat modell itt látható:



24. ábra: Elfogadási munkafolyamat WF-ben

A vezérlőt a következő kódrészlet mutatja be:

```
public class AccController : IAccController
{
    #region IAccController Members
    public event EventHandler<ExternalDataEventArgs>
                                                evtStartWork;
    public event EventHandler<ExternalDataEventArgs>
                                                evtSendToAcceptance;
    public event EventHandler<ExternalDataEventArgs> evtAccept;
    public event EventHandler<ExternalDataEventArgs> evtReject;
    #endregion

    #region IWorkFlowController Members
    public Type StateMachineWorkFlow
    {
        get { return typeof(AccWorkFlow); }
    }
    #endregion
}
```

25. ábra: Munkafolyamat vezérlő

Magát a szolgáltatást pedig az *AccService* osztály implementálja:

```
[StateMachine(typeof(AccController))]
public class AccService : IAccService
{
    // IAccService Members
}
```

26. ábra: Elfogadási szolgáltatás

5.4 A működési infrastruktúra kialakítása

Ebben a fejezetben azt fogom tárgyalni, hogy milyen feltételekre van szükség a szolgáltatás oldalon a rendszer működéséhez, illetve azt, hogy milyen megoldással lehetséges a megszorítások ellenőrzése.

A szerződésben definiált feltételrendszer mellett, a feltételek ellenőrzéséhez a következő információk szükségesek a rendszer működéséhez:

1. A kliens oldali hívás helye (melyik osztályból végezték a hívást)
2. A hívó felhasználói azonosítója
3. A hívó hálózati helye
4. A munkafolyamat aktuális állapota

A szerver oldalon a következő információk állnak rendelkezésünkre:

1. A hívó hálózati helye (ezt az operációs rendszer hálózati alrendszere már lekezeli)
2. A munkafolyamat aktuális állapota

A legtöbb kommunikációs platform, így a WCF is felkonfigurálható úgy, hogy minden egyes híváskor a kliens oldalról a hívó felhasználó adatai automatikusan eljussanak a szerver oldalra. A problémát a kliens oldali hívás helye okozza.

Szerencsére a WCF műveletek kiegészíthetők ún. behaviour-ökkel, „viselkedésmódosítókkal" [108]. Ezek egyszerű metódushívásokat indukálnak az üzleti metódusok meghívása előtt, illetve után mind a kliens, mind a szerver oldalon is. Pontosan erre van szükségünk, ugyanis így azt a kontextust, amelyet a kliens a szerver oldalnak átad, tetszőlegesen módosíthatjuk. Ezen felül a szerver oldalon ellenőrizhetők a megszorítások, valamint sikeres futás után léptethető a munkafolyamat.

A behaviour-ök egy egyszerű XML alapú konfigurációs állomány segítségével hozzáadhatók a rendszerhez [8]. Ez az a funkcionalitás, amely az Aspektus Orientált Paradigma röviden felvázolt fogalomrendszerébe illeszkedik.

Tehát a behaviour-ök a következő műveleteket végzik el:

1. Kliens oldalon a hívás előtt:
    a. A hívási verem felhasználásával megállapítja, hogy honnan történt a hívás
    b. Ezt az információt felveszi a kontextus-paraméterek közé
2. Szerver oldalon a hívás előtt:
    a. Fogadja a kliens oldalról érkező kontextus-paramétereket

b. Ellenőrzi a megszorításokat. Ha van nem teljesülő megszorítás, akkor kivételt dob

3. Szerver oldalon a hívás után:
   a. Amennyiben van munkafolyamat szintű állapotátmenet definiálva az attribútumok között, akkor a sikeres metódushívás esetén elvégzi az állapotátmenetet

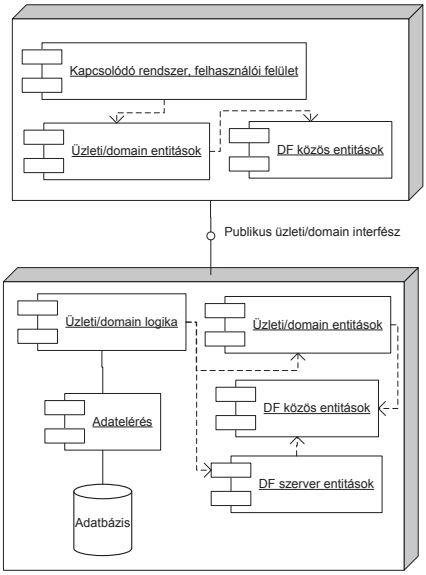
Természetesen az implementáció azt igényli, hogy a szolgáltatások munkamenet (session)[42] [109] azonosítóval dolgozzanak.

5.5 A végleges architektúra

A korábbi fejezetekben ismertettem azokat az alapvető építőköveket, amelyekből a rendszer összeáll. Ebben a fejezetben felvázolom a rendszer architektúráját, azaz a komponenseket és ezek kapcsolódásait, valamint ismertetem, hogy melyik építőkő melyik komponens része.

Az 3.2. fejezetben megmutattam, hogy mi az az alap architekturális felépítés illetve szemléletmód, amely elosztott alkalmazások készítésekor javasolt, sőt elvárt [87].

Tekintsük a következő ábrát:



27. ábra: DF-el kiegészített elosztott rendszer alaparchitektúra

Az ábra az említett fejezetben felrajzolt architektúra ábra kiterjesztése az általam készített elosztott alkalmazások biztonságos fejlesztését támogató keretrendszer komponenseivel (nevük DF-el kezdődik).

Tételesen nézzük végig, hogy melyik komponens milyen építőkövek gazdája:

- Adatelérés
  - Adatbázis-műveleteket megvalósításáért felelős funkciók.
  - Rekordok és mezők megfeleltetése objektumoknak illetve osztályoknak.
- Üzleti/domain entitások
  - Egyszerű, az üzleti szakterületet leíró adattároló osztályok (entitások, domain érték típusok). Ezek az „adatszerződések" a kliens és a szerver között.
  - A publikus üzleti/domain interfészek definíciója. Ezek a szolgáltatás-szerződések a kliens és a szerver között, hogy a szerver milyen publikus funkcionalitásokat ad az őt hívók számára.
  - Ezt a modult mindkét oldalra telepíteni kell
- Üzleti/domain logika
  - A programlogika implementációja legtöbbször szolgáltatások formájában. Ezek a szolgáltatások implementálják az említett publikus interfészt.
  - Ez a komponens felelős közvetlenül, de inkább közvetve a naplózás, a tranzakció-kezelés, a biztonság megvalósításáért.
- Kapcsolódó rendszer/felhasználói felület
  - Ez a kliens logika, amely több fizikai komponensből is állhat.
  - Ezzel a dolgozatban nem foglalkozok
- A DF közös entitások
  - Azok az általam definiált megszorításokat jelző attribútumok, amelyek a kliens oldalon is elhelyezhetők a szerver oldal mellett: StateMachineStep, StateAndPrecondition, AllowedIdentity, AllowedRule, stb.
  - A kliens oldali WCF viselkedésmódosító (behaviour)
- DF szerver entitások
  - A munkafolyamat definíciója, vezérlője
  - Attribútumok, amelyek a szervert és a munkafolyamat típusát összekapcsolják
  - A szerver oldali WCF viselkedésmódosító (behaviour)

5.6 Az elért eredmények összegzése

A 4. fejezetben ismertetett formális modell haszontalan lenne akkor, ha az a gyakorlatban nem segítené a programozók munkáját. A 5. fejezetben egy konkrét implementációt mutattam a formális modellre.

Részleteztem az elérni kívánt célokat, valamint azokat a programozási paradigmákat, amelyek elvei segítségemre lehetnek egy formális rendszer konkrét programozási rendszerre való transzformálása során.

A transzformáció egy egyszerű projekció, átírási rendszer a formalizált világ és a vele barátságban álló deklaratív paradigmát sugalló C# nyelvi attribútumok között.

Ennek a dolgozatnak nem célja az, hogy mély technológiai ismereteket közöljön, ezért a felhasznált .NET alapú technológiákról csak röviden szóltam.

A 4. fejezetben formalizált esettanulmányokat áttranszformáltam egy C# nyelvű megvalósítássá, amelyből egyértelműen látható a módszer működőképessége is.

Ahhoz, hogy a teljes kép kialakulhasson az olvasóban arról, hogy melyik architekturális építőkő milyen szerepben és milyen kapcsolatban áll a többi építőelemmel, a teljes rendszer architektúráját felrajzoltam, majd meghatároztam, hogy melyik elem melyik komponensben kell, hogy helyet foglaljon.

A megvalósítás során több, .NET technológiát felhasználtam, ezek közül a legfontosabb a kommunikációért felelős WCF (Windows Communication Foundation), valamint a munkafolyamatok kezelésért felelős WF (Workflow Foundation).

2. Tézis. A formális modellt felhasználva egy keretrendszert terveztem, amely alapján működő implementációt készítettem szabványos C# nyelven a .NET platformra. Megmutattam egy átírási módszert, amellyel a formális modell által definiált megszorítások egyértelműen átírhatók C# nyelvre. A keretrendszer segítségével megvalósítottam a formálisan is definiált esettanulmányokat.

A témával kapcsolatos eredményeket [8] alatt vázoltam fel.

III. Módszerek .NET programok minőségének javítására

6 Futás idejű napló létrehozása

Mint ahogy az 1. fejezetben említettem, a futás idejű napló a program futása során keletkezett üzenetekből áll. A szakirodalom eddig [42] a következő alapvető kategóriákba sorolta a napló típusait:

1. Üzleti napló
2. Diagnosztikai napló

Az üzleti napló leginkább nagyvállalati (enterprise) rendszereknél használatos. Azokat a bejegyzéseket rögzíti a rendszer működése során, amikor a felhasználók a vállalat számára üzletileg értelmezett műveleteket hajtottak végre (pl. szerződés rögzítése, feladat kiosztása, stb.). A diagnosztikai napló ezzel szemben a rendszer alacsonyabb szintű műveleteit, eseményeit jegyzi fel. Ezek nagyvállalati alkalmazások szintjén egy-egy metódus meghívása, vagy éppen kivétel keletkezése. Továbbá ennek a naplótípusnak fontos szerepe van pl. a szerveralkalmazások körében is. Webkiszolgálók esetében ide kerülnek a felhasználók által meglátogatott oldalak, képek letöltésére vonatkozó információk, vagy éppen egy levelező szerver esetében a fogadott vagy elküldött levelek metaadatai.

Azonban az alkalmazások többségének ennél több információra van szüksége. Célul tűztem ki, hogy egy olyan .NET-közeli naplógenerálási módszert hívjak életre, amely képes egy futó rendszerről olyan részletekbe menő információkat szolgáltatni, amelyek a programozók számára a lehető legtöbb információt megadhatja azok közül, amelyeket a futtatókörnyezet nyilvántart a program állapotaiban. Ezen adatok birtokában pedig hatékonyabbá válhasson a program működése során keletkező hibák, specifikációtól való eltérések felderítése. Mivel az előbbi nem egzakt megfogalmazás, ezért az mondható, hogy a cél egy olyan napló létrehozása, amely legalább a dinamikus szeletelés számára hasznos információkat biztosítja [13].

Tehát a következő bejegyzésekről beszélünk:

1. Melyik változó vett fel értéket (és mit)
2. Melyik változó értékét olvastuk ki (és mi az)
3. Ezek a műveletek milyen utasításban történtek

Kutatásom során két módszert dolgoztam ki futás idejű napló létrehozására:

1. A .NET Debugger API segítségével történő naplógenerálása
2. A .NET Profiler API és IL kód generálás segítségével történő naplógenerálás

Az elsőt alkalmazva a futó programokhoz egy Debuggert csatolhatunk [101],

amely olyan üzeneteket küldhet a programnak, mint lépj egyet, menj el a töréspontig (breakpoint), stb. A futó program minden esetben egy eseménnyel visszajelez az őt felügyelő Debuggernek, a művelet megtörténtéről. Ez a visszajelzés megfelelő mennyiségű információt tartalmaz ahhoz, hogy lokalizálni tudjuk a futtatott programrészt, amely alapján a futás útja rögzítető. A megváltozott illetve olvasott változók értéke szintén megismerhető.

Ezt a módszert [1] írja le. Ott összekapcsoltam a megoldást egy általam implementált alapvető dinamikus szeletelő algoritmussal is.

Mivel a kutatás e fázisában azt tapasztaltam, hogy a napló minősége nem mindig elégséges, valamint a naplókészítés folyamata lassú, illetve a többszálú alkalmazások esetében nem mindig lehetséges, ezért úgy döntöttem, hogy elsődleges célnak a napló készítés minőségének javítását tűzöm ki.

A dinamikus szeletelés kutatása már olyan mértékben előrehaladott, a .NET-programok naplózása pedig még annyira gyerek cipőben jár, hogy egy jobb naplózási módszer kidolgozását tartottam elsődleges feladatomnak.

A keletkezett napló azonban nemcsak az implementációs-tesztelési fázisban, valamint a dinamikus szeletelés esetében alkalmazható. A 7.2. fejezetben további felhasználási módokat, valamint a II. részben felvázolt eredmények integrációját ismertetem.

6.1 A módszer áttekintése

A szakirodalom alapján a következő elvárásokat fogalmaztam meg a kifejlesztendő naplózó metodológiával [2] kapcsolatban:

1. Képes legyen metódusok meghívását, metódusokból való visszatérést rögzíteni.
2. Képes legyen a metódusokban található alapvető utasításokat rögzíteni úgy, hogy abból az eredeti forráskód részletre vissza lehessen hivatkozni.
3. Minden utasításban az olvasott és módosított változók körét rögzítse. Opcionálisan a változó értéke is megismerhető legyen.
4. Megfelelően magas teljesítményt nyújtson.
5. Többszálú alkalmazások esetében is üzemeljen úgy, hogy az esetek nagy részében ugyanazt az eredményt naplózza (lásd 6.6. fejezet).
6. Legyen nem intruzív, azaz ne legyen szükséges az eredeti program forráskódjának módosítása.
7. Az eredmény manuálisan és automatikusan is értelmezhető legyen.

8. A keletkezett napló dinamikus szeletelés bemeneteként felhasználható legyen.

A .NET Profiler API [56] képes a program futása során különböző eseményeket elkapni és tetszőleges kódot futtatni annak hatására. Azt az eseményt készítettem fel a futás idejű napló létrehozására, amikor a Profiler azt jelzi, hogy egy metódus IL kód JIT fordításon fog átesni. Az IL kódba a megfelelő helyekre szondákat helyezek el, amely képes az adott utasítás végrehajtását illetve változó olvasási és írási műveleteket naplózni. Az eredeti IL kód helyett az így módosított IL kódot küldöm tovább JIT fordításra. A CPU tehát már a szondákkal ellátott kódot futtatja le, amelynek hatására naplózásra kerülnek a műveletek.

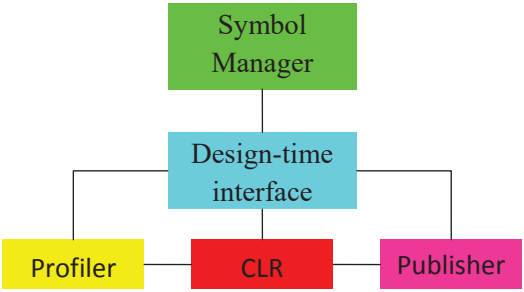
6.2 .NET technológiai áttekintés

Szükséges még az, hogy az olvasót azokkal az alkalmazott .NET eszközökkel és technológiákkal is megismertessem, amelyeket a naplózó eljárás létrehozásakor alkalmaztam.

Először egy magas szintű architekturális áttekintést adok (6.2.1), majd pedig a .NET-metódusok belső reprezentációját fejtem ki (6.2.2).

6.2.1 A .NET Debugging és Profiling infrastruktúra ismertetése

A .NET Debugging és Profiling infrastruktúra [59] komponenseinek kapcsolatait a következő ábra mutatja be:



28. ábra: .NET Debugging és Profiling infrastruktúra

Nem mindegyik komponens rendelkezik magyar szakirodalombeli megnevezéssel, ettől függetlenül mindenhol megadok egy magyar elnevezést is.

A *CLR* (lásd 2. fejezet) fogalmát, szerepét már vázoltam, ezért itt nem ismétlem meg újra.

A *Design-time interface* (tervezési idejű felület) a Debuggerhez kapcsolódik, feladata a Debugger események feldolgozása. Egy olyan, a debuggolt alkalmazástól külön folyamatban fut, ahol egy dedikált szál végzi a Debugger események fogadását. Amikor egy esemény bekövetkezik, akkor a debuggolt alkalmazás megáll és ez a dedikált szál kapja meg a vezérlést. Az esemény feldolgozása után a soron következő Debugger utasítást küldhetjük el az alkalmazásnak, amely bekövetkezése után szintén egy eseményt fog előidézni.

A *Symbol Manager* a lefordított programkód és az eredeti forrásnyelvi program közötti kapcsolat megteremtéséért felelős. A .NET-fordítók által generált fordítási egységekben (assembly) *metaadatok*at is találunk. A metaadatok írják le azt, hogy milyen típusokkal bír az assembly, milyen metódusokat, változókat tartalmaznak a típusok, a metódusok milyen paraméterekkel, visszatérési értékkel rendelkeznek. Látható, hogy ez alapján, az információ alapján az eredeti forrásszöveg és a lefordított kód között nem tudjuk feloldani az összefüggéseket, azaz nem tudjuk meghatározni, hogy melyik IL kódblokk melyik forrás szövegrészből jött létre. Amikor egy programot *Debug módban* fordítunk le, akkor egy *PDB* (Program DataBase) állomány is keletkezik, amelyben a metaadatokban szereplő típus és osztálytag szintű adatok részletesebb információkkal vannak kiegészítve. Ezen felül szekvencia pontonkénti (lásd 6.3. fejezet) kapcsolatokat tárolja a lefordított IL kód és az eredeti forrásszöveg között. A Symbol Manager felelős a PDB állományok kezeléséért és értelmezéséért.

A *Publisher* (kiadó) felelős azért, hogy lekérdezze és felsorolja a .NET folyamatokat.

A *Profiler* szintén az alkalmazások nyomkövetését végzi, azonban itt a futással és teljesítménnyel kapcsolatos eseményeket követhetjük. A Profiler in-process fut, azaz a profilolt alkalmazáson belül. Segítségével olyan eseményeket észlelhetünk, kezelhetünk, mint pl. modul betöltés, osztály betöltés és eltávolítás, metódusonkénti JIT fordítás, metódushívás, kivételek keletkezése és elkapása, vagy éppen a szemétgyűjtő algoritmus futása, stb.

Számunkra a Profiler segítségével történő naplógenerálás során a Profiler és a Symbol Manager tartoznak a szükséges komponensek közé.

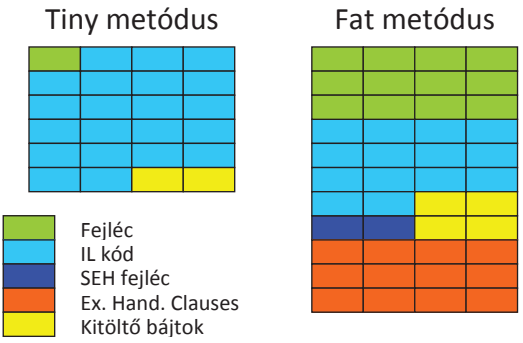
6.2.2 A .NET-es metódusok belső reprezentációja

Ebben a fejezetben ismertetni fogom a különböző metódus típusok felépítését valamint korlátaikat, csoportosítom az IL utasításokat, valamint bemutatom a kivételkezelő záradékot (EHC – Exception Handling Clause) [101].

Minden .NET metódus rendelkezik egy fejléccel, valamint IL utasításokkal. Ezen felül tartalmazhat kitöltő (padding) bájtokat, SEH (Structured Exception

Handling) fejlécet [101] valamint kivételkezelő záradékot.

Egy .NET metódus lehet *Tiny* (kicsi) illetve *Fat* (kövér):



29. ábra: .NET metódusfajták kategorizálása

A *Tiny* metódusokkal szemben felállított követelmények a következők:

1. méretük kisebb 64 bájtnál,
2. a verem nem mélyebb 8 rekesznél,
3. nem rendelkeznek lokális változókkal,
4. nem rendelkeznek SEH fejléccel valamint EHC-vel.

A *Fat* besorolásba kerül minden más metódus, amely egy vagy több feltételt megszeg a fentiek közül.

Az IL utasítások egymás között nem regiszterek segítségével kommunikálnak, hanem egy közös vermen keresztül. Ez azt jelenti, hogy a művelet eredménye a veremre kerül, illetve a veremről vehetnek le értékeket a műveletek elvégzéséhez. Emellett lehetőség van paraméterek megadására is. Természetesen a JIT-fordításon átesett IL utasításokból keletkezett gépi kód már optimálisan kihasználja a hardverplatform lehetőségeit, és ahol csak lehet, regisztereket használ az adatok kezelése során.

Az IL utasítások kategóriákba sorolhatók a paramétereik száma és típusa alapján [81][85]. Ezt a besorolási rendszert a következő táblázatban foglalom össze:

Paraméterek karakterisztikája	Magyarázat, példák
Paraméter nélküli IL utasítások	Olyan utasítások, amelyek csak a veremről származó adatokat használnak fel a műveletvégzéshez. Pl. 7. dup: duplikálja a vermen található legfelső értéket 8. ldc.i4.-1,…,ldc.i4.8: ezek az utasítások gyakran használt 32 bites egész számok (-1, 0, 1, …, 8) valamelyikét töltik a veremre 9. add: a két legfelső veremértéket összeadja, leveszi a veremről, majd az eredményt a veremre teszi
Egy egész szám paramétert váró utasítások	Ezek az utasítások a paraméter érték alapján műveletet végeznek a vermen, vagy éppen a vezérlést átadják a paraméterrel specifikált helyre (függvényen belül). A szám 8, 16, 32, illetve 64 bit széles lehet. Pl. 10. ldc.i4 <int>: az <int> értékű 32 bites egész számot tölti a veremre 11. br <reloff>, br.s <reloff>: rövid vagy hosszú ugrást hajtanak végre az utasítások a paraméterben megadott, az ugró utasítás helyéhez képest relatív címre. A hosszú ugrás 32 bites egész számot vár, a rövid ugrás pedig 8 bites egész számot vár.
Token érték paramétert váró utasítások	Minden típust és típus tagot token értékkel azonosít a keretrendszer. A token végső soron egy 32 bites szám, amely kitüntetett szereppel rendelkezik. A tokenek értéke fordítási időben dől el, biztosítva ezzel egyediségüket. Olyan IL utasítások tartoznak ide, amelyek valamilyen típussal, tagmetódussal, tagváltozóval dolgoznak. Pl. 12. call <token>: A token értékkel megadott függvényt hívja meg. Természetesen nem statikus metódusok esetében szükséges, hogy a verem legfelső értéke azt a példányhivatkozást tartalmazza, amelyen a függvényt hívni kívánjuk. 13. box <token>: A verem tetején található <token>

	érték típusú értéket becsomagolja (boxing) egy object-be. 14. ldfld <token>: a verem tetején található objektum <token> -el azonosított mezőjét tölti be a veremre
Többparaméteres utasítások	Több paramétert is feldolgozni képes IL utasítások tartoznak ide. Ezekből van a legkevesebb. Az első paraméter mindig az őt követő további paraméterek darabszámát jelöli. Pl. 15. switch <count><reloff_1>…<reloff_count>: A verem tetején egy index található, amely alapján kiválasztásra kerül az a relatív cím, ahova a vezérlés átkerül. Gyakorlatilag ez az utasítás a magas szintű C# nyelvi switch utasításnak felel meg.

30. ábra: IL utasítások kategorizálása

Az EHC (kivételkezelő záradék) ismertetése sem maradhat el. Tudjuk, hogy minden *Fat*, azaz kövér metódus egy vagy több kivételkezelő *try-catch* blokkot foglalhat magába. Az EHC feladata az, hogy tárolja ezeknek a blokkoknak az IL szekvencia szintű elhelyezkedését és hosszát, valamint azt, hogy milyen típusú kivételek kezelését végzi a *catch* blokk. Az EHC, a metódusoknál bevezetett terminológia szerint szintén lehet *Tiny* illetve *Fat* formátumban. A *Tiny* EHC esetében a *try* és *catch* blokkok elhelyezkedése kétbájtos formátumban van specifikálva, míg hosszuk egy bájtos formátumban, mialatt a *Fat* EHC esetében minden paraméter négy bájt széles.

6.3 Szekvencia pont szintű futás idejű naplógenerálás

A *szekvencia pont* egy programozási nyelvek által definiált fogalom [57], amely azt határozza meg, hogy ha a fordító szekvencia ponthoz ér, akkor ott be kell fejeznie a kifejezések kiértékelését. Másként megfogalmazva a szekvencia pont egy olyan pont a futás során, amely azt biztosítja, hogy minden azt megelőző utasítás lefutott és a hatása érvényesül a futás ezt követő utasításaira. Forrásszöveg szinten ezt legtöbbször a ; illetve a , karakterek jelzik.

A szekvencia pont szintű naplógenerálás alatt a következőt értem: azt naplózzuk, hogy a program milyen utasításokat hajtott végre futás közben, valamint az itt keletkező információkat meg tudjuk feleltetni az eredeti

forráskódban található utasításoknak.

A .NET Profiler egy *ICoreProfilerCallback2* COM [35] interfészt ad, amely egy COM osztályként implementálható. Az implementáció nem készülhet menedzselt környezetben, ezért a C++ nyelvet [66] választottam.

A metaadatok és a szimbólumok feloldásához pedig a következő interfészek implementációit használtam fel: *ISymUnmanagedReader*, *ISymUnmanagedMethod*, *IMetaDataImport* valamint *ICorProfilerInfo2*.

A több mint 70 eseményből, amelyet az ICorProfilerCallback2 interfész definiál, mindösszesen kettőt használtam: *ModuleLoadFinished* és *ClassLoadFinished*.

6.3.1 Naplózó metódusok – implementáció és ráhivatkozás

A naplózó metódusok feladata az, hogy a paraméterül kapott információkat a naplóba írják. Ezekre a külső modulban elhelyezett naplózó metódusokra majd futási időben fogunk hivatkozásokat adni, amelyek segítségével meghívhatók.

Ebben a fejezetben azt fogom megmutatni, hogy milyen naplóbejegyzés író metódusokat készítettem valamint azt, hogy ezeket milyen módon lehetséges használatba venni.

Egy *TracerModule* nevű modult hoztam létre, amelynek a *Tracer* nevű statikus osztályában helyeztem el két statikus metódust *DoFunc* illetve *DoTrace* néven. Az előbbi feladata, hogy a függvény belépéseket illetve kilépéseket naplózza, az utóbbi pedig a függvényen belüli szekvencia pontok elérését jelzi.

A DoFunc metódus implementációja a következő:

```
public static void DoFunc(uint startLine, uint startColumn,
                          uint endLine, uint endColumn,
                          uint functionID, uint action)
{
  try
  {
    lock (lockObj)
    {
      char act = 'E';
      if (action == 2)
      {
        act = 'L';
      }
      sw.WriteLine("{6}T{5}{4}{0}:{1}-{2}:{3}", new object[] {
                  startLine, startColumn,
                  endLine, endColumn, act, functionID,
                  Thread.CurrentThread.ManagedThreadId });
    }
  }
  catch
  {
  }
}
```

31. ábra: Metódus belépés/kilépés naplózása

A naplózó eljárásokkal szemben támasztott alapvető követelmény, hogy ha hibára futnak, akkor nem ránthatják magukkal a futó alkalmazást, ezért egy try-catch blokkot alkalmaztam.

Felkészítettem a megoldást többszálú környezetben való futásra is, ezért a *lock* kulcsszó segítségével egy kritikus szekciót (Critical Section) hoztam létre. Az *action* paraméter értéke határozza meg, hogy *E(=Enter=Belépés)* vagy *L(=Leave=Kilépés)* történt-e a nyomkövetett metódusba/ból. Egy *stream*re, folyamra soronként írom ki a naplót, ez esetben a stream egy fájlba küldi a kimenetet. A stream természetesen tetszőleges helyre és módon jegyezheti le a naplóbejegyzéseket. Kiírásra kerül a szekvencia pont (ez a függvény elején vagy végén van) kezdetére illetve végére mutató sor- és oszlopindexe, a belépés/kilépés ténye, a metódus egyedi azonosítója, valamint a szál egyedi azonosítója.

A *DoTrace* metódus implementációja a következőképpen fest:

```
public static void DoTrace(uint startLine, uint startColumn,
                           uint endLine, uint endColumn,
                           uint functionId)
{
  try
  {
    lock (lockObj)
    {
      sw.WriteLine("{5}T{4}I{0}:{1}-{2}:{3}", new object[] {
                  startLine, startColumn,
                  endLine, endColumn, functionId,
                  Thread.CurrentThread.ManagedThreadId });
    }
  }
  catch
  {
  }
}
```

32. ábra: Szekvencia pont szintű naplózó eljárás

Láthatjuk, hogy ez a metódus egyszerűbb, mert ebben az esetben elég csak a szekvencia pontra vonatkozó adatokat és a szál egyedi azonosítóját rögzíteni. Ezen kívül eltároljuk a függvény egyedi azonosítóját, valamint egy I kódot is, ami ezt jelzi, hogy a függvény belsejében (intra) járunk. Gondolhatnánk, hogy felesleges a függvény egyedi azonosítóját itt is tárolni, erre azonban azért van szükség, mert ha egy metódus kivételt dob, és azt nem kezeljük le, akkor nem kerül a naplóba a metódus elhagyását jelző üzenet. Azaz nem tudnánk, hogy a hiba után melyik függvényben folytatódik a futás, ugyanis a sor és oszlopindex kevés információ egy komplex alkalmazás esetében.

A WriteLine metódusban elhelyezkedő formázó kifejezésben található *{q}* formátumú 0-val kezdődő hivatkozások a *q.* paraméterre hivatkoznak.

Ahhoz, hogy a *TracerModule*-ban fellelhető függvényeket meg tudjuk hívni, a hívó assembly-nek hivatkoznia kell a *TracerModule* assembly-re, illetve minden

egyes használni kívánt típusára és függvényére. Mivel a tervezés során alapkövetelményként állítottam fel, hogy nem módosíthatjuk az eredeti, nyomon követett programot, ezért a behivatkozást a vizsgálandó modulok betöltésekor kell elvégeznünk a memóriában található metaadatokon.

Ez gyakorlatilag azt jelenti, hogy a *ModuleLoadFinished* esemény lefutásakor a metaadatok kiegészítésre kerülnek az előbb említett hivatkozásokkal.

6.3.2 IL kód újraírás

Célom az, hogy még azelőtt módosítsam a metódusok IL kódját, mielőtt azok JIT fordításra kerülnének [56]. A *ClassLoadFinished* profiler eseményt választottam ki ennek a műveletnek az elvégzésére, mivel ebben a korai szakaszban, eseményben az osztály minden metódusát enumerálhatom, majd egy lépésben módosíthatom az összes metódus IL kódját. A módosított IL kód előállítása után memóriát allokálok az új IL kód bináris reprezentációja számára, ahova azt elhelyezem.

Tehát egy metódus esetében a következő öt lépést hajtom végre azért, hogy a nyomkövető szondák üzemelhessenek:

1. A metódus bináris formátumú IL kódjának értelmezése, saját adatstruktúrában való elhelyezése
2. A metódus és az IL utasítások formátumának felkészítése az IL kód újraírására
3. Az instrumentáló szondák elhelyezése az IL utasításfolyamban
4. Relatív és abszolút címek újraszámítása
5. Az újraírt IL kód eltárolása bináris formátumban

A metódus bináris formátumú IL kódjának értelmezése során először azonosítom a szekvencia pontok által határolt kódblokkok IL bináris szekvencia (bájt tömb) és forráskód szintű kezdő és végpontjait.

A bináris szekvenciát most már a szekvencia pontok által határolt blokkokra darabolhatom szét, majd elkezdhető az IL utasítások visszafejtése.

Tekintsük a következő metódust:

```
static bool IsFirstLess(int value1, int value2)
{
  if (value1 < value2)
  {
    Console.WriteLine("Yes, first is less");
    return true;
  }
  return false;
}
```

33. ábra: Példa metódus C# kódja

A metódus nagyon egyszerű feladatot hajt végre: a paraméterül kapott két egész számról eldönti, hogy az első kisebb-e, mint a második. Ha igen, akkor ezt a képernyőre is kiírja.

A következő táblázatban azt mutatom be, hogy a fenti metódus milyen szekvencia pontok által határolt blokkokból áll, azok hol kezdődnek az IL bináris szekvencia szintjén valamint hol kezdődnek és végződnek forráskód szinten. A forráskód szintű elhelyezkedés esetében a sor és oszlop indexeket vesszővel választom el egymástól.

Sorszám	IL eltolás	Kezdőpont a forráskódban	Végpont a forráskódban
0	0	25,1	25,2
1	1	26,3	26,23
2	9	0xfeefee,0	0xfeefee,0
3	12	27,3	27,4
4	13	28,7	28,47
5	24	29,7	29,19
6	28	31,3	31,16
7	32	32,1	32,2

5. táblázat: Példa metódus szekvencia pontjai

A 2. sorszámú szekvencia pont nem rendelkezik forráskód szintű elhelyezkedéssel. Ennek a ténynek a belső reprezentációja egy Feefee-nek (Fifi) becézett sorszámmal jelölődik.

Miután meghatároztam, hogy milyen szekvencia pontok által határolt blokkokra osztható fel a metódus, következhet az IL utasítások visszafejtése, meghatározása. Minden IL utasítás bináris reprezentációjának formátuma része a .NET szabvány [81][85] dokumentációjának. Ennek a dokumentumnak a felhasználásával készítettem egy olyan értelmezőt, amely a bájtfolyamot feldolgozva egy IL utasításlistát hoz létre.

A következő lista a fenti metódus ezzel a módszerrel visszafejtett IL utasításait tartalmazza:

```
0: nop                                  18: call 167772181
1: ldarg 0                              23: nop
2: ldarg 1                              24: ldc.i4 1
3: clt                                  25: stloc 0
5: ldc.i4 0                             26: br.s 32 (4)[tsp: 7,til: 0]
6: ceq                                  28: ldc.i4 0
8: stloc 1                              29: stloc 0
9: ldloc 1                              30: br.s 32 (0)[tsp: 7,til: 0]
10: brtrue.s 28(16)[tsp:6,til:0] 32: ldloc 0
12: nop                                 33: ret
13: ldstr 1879048193
```

34. ábra: Példa metódus IL kódja

Az utasítások előtti számok az IL eltolás értékeket mutatják (ez látható a fenti táblázatban is). Jelen dolgozatnak nem célja az IL utasítások részletes ismertetése, ezért a szabvány tanulmányozását ajánlom az olvasó figyelmébe**.** Az ugró, vezérlésátadó utasítások igényelnek egy kis magyarázatot (a fenti példában a *brtrue.s* illetve a *br.s* utasítások). Ezek az utasítások belső reprezentációjuk szerint, relatív címekkel határozzák meg az ugrás helyét, azonban a feldolgozás során ennél többre van szükségünk. Később rávilágítok ennek okára. Az ugró utasítások után szereplő első számjegy az abszolút címet jelöli, majd zárójelben szerepel a relatív eltolás, amely mindig az ugró utasítást követő bájt indexéhez képest van megadva. A szögletes zárójelben a *tsp* illetve a *til* értékek a már feloldott cél szekvencia pont sorszámot és az azutáni eltolást jelzik.

A metódus és az IL utasítások formátumának felkészítése az IL kód újraírásra lépés szerepe a következő: mivel extra IL utasításokat fogunk elhelyezni a metódus kódjában, ezért *Tiny* metódusok (lásd 6.2.2. fejezet) esetében túlléphetjük az ezzel járó 64 bájtos méretkorlátot, illetve a rövid ugró utasítások (pl. *br.s*, *brfalse.s*, *brtrue.s*, stb.) esetében az 1 bájton ábrázolt relatív címtartomány által meghatározott maximális ugráshosszt.

A *Tiny* metódus formátumot *Fat* formátumra módosítom a metódus metaadatainak átírásával. Ezek után minden rövid ugró utasítást megfeleltetünk a hosszú párjának. Ezek alapján a *br.s*-ből *br* lesz, a *brtrue.s*-ből *brtrue* lesz, a *brfalse.s*-ből *brfalse* válik, vagy éppen a *bge.un.s*-t *bge.un* utasítássá módosítom, stb. Erre a feladatra az Objektum orientált [55] programfejlesztésben használatos visitor (látogató) [37] tervezési minta a legmegfelelőbb.

Fontos, hogy a *Tiny* típusú kivételkezelő zárvány blokkokat szintén *Fat* formátumúra bővítsük, mert az itt fennálló korlátozásokat könnyen túllépheti a metódus, ha extra utasításokat helyezünk el benne.

Az instrumentáló szondák elhelyezése az IL utasításfolyamban lépés végzi a fő feladatot. Az instrumentálás kifejezés az angol eredetű instrumentation azaz felhangszerelés kifejezésből származik.

Addigra, amikore erre a pontra ér a feldolgozás a következők állnak rendelkezésre, az alábbiak történtek meg:

1. A nyomkövető metódusok token értékeit megismertük már a fordítási egység (assembly) betöltésekor, metaadatainak módosításakor.
2. A metódusok bináris kódját értelmeztük, IL kóddá alakítottuk.
3. A metódusok, ugró utasítások, kivételkezelő zárványok kibővítése megtörtént.

Mindezek után már rátérhetek arra, hogyan lehetséges az instrumentáló

szondák elhelyezése. A szekvencia pont szintű instrumentáláskor használatos szondákat két kategóriába soroltam:

1. Metódus belépéskor, illetve metódusból történő visszalépéskor alkalmazott szonda.
2. Belső szekvencia pontoknál elhelyezendő szonda.

A megoldásomat az első esetre vetítve mutatom be, ugyanis a második eset az ebből következik.

Egy bájt tömböt hozok létre, amely olyan IL utasításokat tartalmaz, amelyek az instrumentáló metódus paraméterezését és meghívását végzi. Ez a reprezentáció könnyen integrálható az eddig elkészültekkel. Tekintsük a következő C++ nyelvű kódrészletet:

```
BYTE insertFuncInst[31];
insertFuncInst[0] = 0x20;  //ldc.i4, start line
insertFuncInst[5] = 0x20;  //ldc.i4, start column
insertFuncInst[10] = 0x20; //ldc.i4, end line
insertFuncInst[15] = 0x20; //ldc.i4, end column
insertFuncInst[20] = 0x20; // ldc.i4, func. Id
insertFuncInst[25] = 0x0;  // ldc.i4.1 or ldc.i4.2
insertFuncInst[26] = 0x28; // call
*((DWORD *)(insertFuncInst+27)) = tracerDoFuncMethodTokenID;
```

35. ábra: Instrumentáló szonda sablon

A metódus első 5 paraméterét az ldc.i4 (0x20 és egy 4 bájtos egész szám érték) utasítással, majd pedig a 6. konstans értéket betöltő ldc.i4.1 (belépés = 1) vagy ldc.i4.2 (kilépés = 2) utasítással adjuk meg. Ezután már meghívható a nyomkövető metódus (lásd 6.3.1).

A hiányzó paraméterértékeket dinamikusan behelyettesítem minden egyes belépéskor illetve kilépéskor, illetve a hasonló behívási megoldást alkalmazó belső szekvencia pontok esetében is.

Ezeknek a végső soron binárisan ábrázolt IL utasítás-szekvencia sablonként működő konkrét értékekkel behelyettesített bájt tömböknek az értelmezése után ugyanolyan IL utasítás szekvenciákat kapunk, mint amilyenné magát a módosítás alatt álló metódust is alakítottam. A kettőt összevezetem, azaz a metódus IL szekvenciájának megfelelő pontjaiba beszúrom a szondák IL kódját.

A **relatív és abszolút címek újraszámítása** ezentúl már elvégezhető. Ez azért szükséges, mivel az IL szekvencia módosítása során a (eredeti programra vonatkozó) relatív és abszolút címek érvénytelenné váltak. A feladat következésképpen az, hogy ezeket a címeket úgy módosítsuk, hogy a program funkcionalitása megfeleljen a régi programénak. Szükséges tehát az ugró utasítások relatív címeinek módosítása, illetve a kivételkezelő zárványok által

meghatározott címek módosítása.

Az ugró utasítások nem feltétlenül egy szekvencia pont után következő legelső IL utasítására adják a vezérlést, azaz előfordul az is, hogy egy belső utasítást címeznek meg. Erre azért szükséges felhívni a figyelmet, mivel az instrumentáló szondákat mindig a szekvencia pont után közvetlenül helyezem el. Ha az eredeti ugró utasítás a szekvencia pont után következő legelső utasításra adja a vezérlést, akkor úgy kell módosítani az ugró utasítások célját, hogy azok a szonda legelső utasítására adják a vezérlést, azaz ugrás után is nyomon lehessen követni a végrehajtási utat a program futása során. Ha nem egy, közvetlenül a szekvencia pont után következő utasításra adta a vezérlést egy ugró utasítás, akkor a módosított IL szekvenciában erre az utasításra kell adni a vezérlést a szonda helyett. Ez az eset például akkor fordulhat elő, amikor egy bonyolultabb logikai kifejezést tartalmazó feltétel (if) kiértékelésére kerül sor.

A cél címek könnyen meghatározhatók, ugyanis a belső reprezentációmban szereplő minden IL utasítás vissza tudja adni a hosszát.

Az EHC esetében történő címszámítás ehhez hasonlóan végezhető, ezért külön nem térek ki rá.

Az összes előző lépésen, illetve transzformáción átesett *IsFirstLess* metódus esetében a következő IL szekvencia fog létrejönni:

```
0: ldc.i4 25          71: ldc.i4 27         159: stloc 0
5: ldc.i4 1           76: ldc.i4 3          160: br 197 (32)
10: ldc.i4 25         81: ldc.i4 27         165: ldc.i4 31
15: ldc.i4 2          86: ldc.i4 4          170: ldc.i4 3
20: ldc.i4 3          91: call 167772194    175: ldc.i4 31
25: ldc.i4 1          96: nop               180: ldc.i4 16
26: call 167772195    97: ldc.i4 28         185: call 167772194
31: nop               102: ldc.i4 7         190: ldc.i4 0
32: ldc.i4 26         107: ldc.i4 28        191: stloc 0
37: ldc.i4 3          112: ldc.i4 47        192: br 197 (0)
42: ldc.i4 26         117: call 167772194   197: ldc.i4 32
47: ldc.i4 23         122: ldstr 879048193  202: ldc.i4 1
52: call 167772194    127: call 167772181   207: ldc.i4 32
57: ldarg 0           132: nop              212: ldc.i4 2
58: ldarg 1           133: ldc.i4 29        217: ldc.i4 3
59: clt               138: ldc.i4 7         222: ldc.i4 2
61: ldc.i4 0          143: ldc.i4 29        223: call 167772195
62: ceq               148: ldc.i4 19        228: ldloc 0
64: stloc 1           153: call 167772194   229: ret
65: ldloc 1           158: ldc.i4 1
66: brtrue 165(94)
```

36. ábra: Instrumentált IL kód

A szondák kódját dőlt piros karakterekkel emeltem ki.

Az újraírt IL kódot most már eltárolhatjuk bináris formátumban. Birtokában vagyunk a módosított IL szekvenciának a saját adatstruktúránkban, amelyet átalakíthatunk bináris reprezentációvá. Minden metódus esetében a bináris, bájt tömbben ábrázolt kódrészlet számára memóriát foglalok, és tudatom a futtatókörnyezettel, hogy mostantól ezt tekintse a metódus kódjának.

6.4 Változó szintű futás idejű naplógenerálás

A szekvencia pont szintű futás idejű naplógenerálás, amelyet a 6.3-ban ismertettem, már rengeteg információval segíti a futás idejű hibák felderítésének hosszú és nehézkes folyamatát, ezáltal a minőség javítását. Innentől kezdve megismerhetővé válik az, hogy milyen futási útvonalat járt be a program. Megszerezhető információ továbbá az, hogy mikor történt szál kontextus váltás (thread context switch), azaz mikor adta át az operációs rendszer ütemezője egyik szálról a vezérlést egy másikra. Ezen felül lehetőség van egy olyan szeletelő algoritmus kifejlesztésére, amely a statikus programszeletelés során létrehozott program szeleteket képes lecsökkenteni azokra a programutasításokra, amelyek ténylegesen futottak.

A módszer kifejlesztése és eredményeinek ismerete megnyitotta az utat a bővebb napló létrehozása felé. Ha ismernénk, hogy a futás során melyik változó került kiolvasásra vagy kapott értéket, ezen felül az értéket vagy annak egy részét is megismernénk, akkor sokkal több információval rendelkeznénk a futás során történtekről. Birtokában lennénk, hogy melyik változó milyen értéket vett fel, tudnánk, hogy a változók mikor kaptak helytelen értéket. De ami még fontosabb, hogy a dinamikus szeletelés számára is megfelelő részletességű napló jönne létre.

Ezek után kutatásom következő állomásának a változó szintű futás idejű naplógenerálás életre hívását jelöltem ki, amelynek segítségével a változók kiolvasását és definiálását (írását, módosítását) naplózhatjuk [9].

Ha megnézzük a 6.3.2-ban lefektetett 5 lépést, akkor azt mondható, hogy a 3. és a 4. lépés közé beékelhető egy újabb lépés, amely a változók nyomkövetését is képes lenne előkészíteni.

Ebben a fejezetben naplózási, naplózhatósági szempontból csoportosítom a változókat, majd tárgyalom a változók illetve a változók típusának felderítését, meghatározásának módját, végül pedig az instrumentációs kód elhelyezését veszem górcső alá.

6.4.1 Változó kategóriák

A változókat a következő tulajdonságaik alapján kategorizálom:

1. A változó deklarálásának helye.
2. Érték vagy referencia típusú változóról beszélünk.
3. A változó „rendszer osztálya”.

Egy változó a deklarálás helye (1) szerint lehet lokális változó (amelyet egy metóduson belül definiálunk és használunk), lehet egy metódus paramétere, vagy egy osztály tagváltozója is.

A .NET specifikáció két fő kategóriába osztja a változókat: érték és referencia típusok (2). Az érték típusokat a vermen tároljuk, ezek nem vehetnek fel *null* értéket, a szemétgyűjtő algoritmus nem foglalkozik velük, valamint az értékadás operátor egy másolatot készít a változóból az értékadás során. Ebbe a kategóriába tartozik a legtöbb egyszerű típus (szám típusok: byte, short, int, char, float, bool, stb.) valamint a *struct*. A referencia típusok a halomterületen (heap) tárolódnak, felvehetnek *null* értéket, a szemétgyűjtő algoritmus felelős az általuk használt memória felszabadításáért, valamint az értékadó operátor nem a teljes tartalmukat másolja le, hanem csak egy hivatkozást (referencia) a tartalomra. Ide tartozik pl. a string (immutable is [57]) és minden *class* (osztály). Létezik egy harmadik kategória is a *nullozható* (*nullable*), amely olyan érték típusokat takar, amelyek felvehetnek *null* értéket is.

A változó „rendszer osztálya” (3) fogalom alatt azt értem, hogy milyen mélyen integrált a változó típusa a .NET keretrendszerrel, azaz mi a típus reprezentációs módja a metaadatokban. Az elemi, egyszerű típusokat, mint pl. az int, string, object, stb. a metaadatokban egy bájtos formátumban jelöljük, míg az összetett típusokat, mint pl. a struktúrák és osztályok a típus token értékük alapján reprezentáltak.

6.4.2 Változók és a változók típusának felderítése

Szükséges az, hogy ismerjük a változók típusát, valamint, hogy azt is naplózni tudjuk, hogy mi a változó deklarálási helye, illetve érték vagy referencia típusról beszélünk-e.

Amikor majd futás időben a változó olvasását vagy definiálását követő szonda metódus meghívásra kerül, akkor ennek egyik paramétere nyilvánvalóan a változó kell, hogy legyen. Nem kívánok, és nem is lehetséges minden változótípushoz külön naplózó metódust készíteni, ezért minden esetben object típusú változót várunk, amely minden .NET-es típus őse. Ekkor azonban probléma merülhet fel a típuskonverzióval. A referencia típusú változók esetében

könnyű a metódus meghívása, ugyanis a referencia típusú változók automatikusan kasztolódnak object-re. Az érték típusú változók esetében azonban szükséges egy explicit dobozolás (boxing) elvégzése. A *box* IL utasítás paraméterül várja a dobozolni kívánt változó típus token értékét, amit az érték megismeréséhez enumerálni, felderíteni szükséges.

A változók és a változók típusának felderítése, meghatározása egy két fázisú folyamat. Az első akkor történik, amikor egy assembly betöltésre kerül. Ebben a szakaszban a definiált és más modulokból behivatkozott típusok (osztály, struktúrák, egyszerű típusok) és típus tagváltozók kerülnek felderítésre. A második fázis az osztálybetöltéskor történik meg, ekkor ugyanis a lokális változókat és metódus paramétereket enumerálja a megoldásom.

Mint már korábban említettem, az egyszerű típusokat egy bájt segítségével azonosítja az assembly metaadata. Ezen felül ezek a változók ugyanúgy hivatkozásra kerülnek típus token segítségével is (*mscorlib*-ből), mint a többi komplex változó [81][85].

A változók típusának alacsony szintű felderítése sajnos nem lehetséges az általunk használt COM API segítségével. A változó típusinformációi egy ún. *signature blob* bináris csomagban találhatók meg. A COM API határai addig tartanak, amíg általa lekérdezhető ez a bináris adatcsomag. A csomag értelmezése megoldandó feladatként jelentkezik. Az implementáció során David Broman CLR Profiling API nevű blogján [93] található signature blob értelmezőt használtam fel, szabtam testre és integráltam a megoldásommal.

Most már minden információ rendelkezésünkre áll ahhoz, hogy a változó szintű szondákat elhelyezzük az IL utasítás szekvenciában.

6.4.3 Változó nyomkövető szondák elhelyezése

Összesen 6 különböző instrumentáló metódust (szondát) különböztettem meg, amelyeket a 6.3.1-ben vázolt osztályban helyeztem el, mint statikus metódusok. A 6 metódus 6 különböző változó használatot naplóz:

1. lokális változó definíció,
2. lokális változó felhasználás,
3. metódus paraméter definíció,
4. metódus paraméter felhasználás,
5. osztály tagváltozó definíció,
6. osztály tagváltozó felhasználás.

Nem szükséges, hogy tovább bontsam ezt a kategorizálást a 6.4.2 alapján, ugyanis *object* típusú paraméterként átadható referencia és érték típusú, elemi és

komplex besorolású változó is.

Meg kell határozni azokat a helyeket az IL utasítás szekvenciában, ahova a változó felhasználást, illetve a definíciót naplózó szondákat el kell helyezni. Minden olyan utasítást azonosítottam, amely egy változó értékét tölti a veremre vagy éppen a verem legfelső elemét írja a változóba. Ezek az utasítások a következők:

1. *ldloc*, *ldloc.s*: lokális változó betöltése a veremre – lokális változó felhasználása
2. *stloc*, *stloc.s*: verem legfelső értékének lokális változóba tétele – lokális változó definiálás
3. *ldarg*, *ldarg.s*: metódus paraméter betöltése a veremre – paraméter változó felhasználása
4. *starg*, *starg.s*: verem legfelső értékének paraméter változóba tétele – paraméter változó definiálás
5. *ldfld*: osztály tagváltozó betöltése a veremre – osztály tagváltozó felhasználása
6. *stfld*: verem legfelső értékének osztály tagváltozóba tétele – osztály tagváltozó definiálása

A szondát minden változó felhasználó utasítás után (azaz amikor már a vermen van), illetve minden változó definíciós utasítás előtt (azaz amikor még a vermen van) elhelyezzük.

Mivel a szonda felhasznál egy változó példányt, azaz leveszi a veremről, ezért az szükséges, hogy a verem legfelső elemét duplikáljuk az e célt szolgáló *dup* IL utasítás felhasználásával. További előkészítő lépéseket kell tenni:

1. az érték típusokat dobozolni kell,
2. a paraméter és a lokális változók esetében az index (hányadik paraméter, hányadik lokális változó) értéket is szükséges letárolni.

A következő C++ nyelvű kódrészlet egy olyan hívási sablont mutat, amely lokális érték típusú változó felhasználásának naplózását végzi:

```
BYTE insertTraceLocalUseValueInst[16];
insertTraceLocalUseValueInst[0] = 0x25; //dup
insertTraceLocalUseValueInst[1] = 0x8c; //box
insertTraceLocalUseValueInst[6] = 0x20; //ldc.i4
insertTraceLocalUseValueInst[11] = 0x28; // call
*((DWORD *)(insertTraceLocalUseValueInst+12)) =
                        tracerDoLocalVarUseMethodTokenID;
```

37. ábra: Változó szintű szonda sablon

Több paramétert dinamikusan helyettesítek be. A *box* utasítás a 6.4.2–ben ismertetettek alapján a forrás változó típusának token értékét várja paraméterül (a sablon 2-5 indexén található 4 bájtos érték). Az *ldc.i4* utasítás pedig a lokális változó indexét várja, ennek szintén kihagytunk helyet.

A lokális referencia típusú változók felhasználásának naplózásának előkészítése annyiban különbözik a fentitől, hogy ott nincs szükség dobozolásra, azaz nincs *box* utasítás. További szonda sablon kódot nem ismertetek, mivel azok könnyen levezethetők az eddig ismertetett megoldásból.

Az érték és referencia típusú lokális változó felhasználásának naplózását végző metódus kódja a következő:

```
public static void DoLocalVarUse(object var, uint index)
{
  try
  {
    lock (lockObj)
    {
      if(var != null)
      {
        Type t = var.GetType();
        if (t.IsValueType)
        {
          sw.WriteLine("{3}LUV{0}:{1}:{2}", index, t,
                       var.ToString(),
                       Thread.CurrentThread.ManagedThreadId);
        }
        else
        {
          sw.WriteLine("{4}LU{2}R{0}:{1}:{3}", index, t,
                       System.Runtime.CompilerServices.
                       RuntimeHelpers.GetHashCode(var),
                       var.ToString(),
                       Thread.CurrentThread.ManagedThreadId);
        }
      }
      else
      {
        sw.WriteLine("{1}LU{0}NULL", index,
                     Thread.CurrentThread.ManagedThreadId);
      }
    }
  }
  catch { }
}
```

38. ábra: Lokális változó használatát naplózó metódus

A változók elérését naplózó eljárásokat is felkészítettem többszálú működésre, azaz kritikus szakaszokat alakítottam ki, valamint minden naplóbejegyzés tartalmazza a szál egyedi azonosítóját is. Megkülönböztetem a *null* és a nem *null*-értékű, valamint az érték és a referencia típusú változókat is. Minden esetben naplózzuk a változó indexet illetve az egyedi szál azonosítót. Azonban a futásidejű típust csak nem *null* érték felvétele esetén van lehetőségünk naplózni, ugyanis a *null* érték nem hordoz magában típusinformációt. A fordítási idejű típus ebben az esetben sem kerül a naplóba, ugyanis ez az információ a forrásszövegből megismerhető. A *ToString* tagfüggvény egyszerű típusok esetében általában a változó értékét adja vissza, komplexebb változók esetében a változó típusát vagy pedig a fejlesztő által definiált értéket szolgáltatja. Amennyiben a *ToString* valamely típus esetében nem mellékhatás-mentes, akkor a típus esetében hagyjuk el a *ToString* meghívását!

Referencia típusú változók esetében hasznos lehet a hash (hasító) [62] kód ismerete is. A hash kód itt egy olyan 32 bites egész szám, amely típusonként csoportosítva ugyanazt az értéket felvevő változók esetében egyedinek kell lennie. Azonban nem feltétel (hiszen nem egyértelműen lehetséges) az injektivitás, azaz hogy eltérő érték esetében a hash kód különbözzön.

Ennek megfelelően igaz az, hogy két ugyanolyan értéket felvevő egész szám esetében a hash kódok megegyeznek, eltérő számok esetében különböznek. Például két ugyanazt a szöveget felvevő *string* típusú változó esetében a hash értékek azonosak, azonban két különböző szöveg is adhatja ugyanazt a hash kódot. A *GetHashCode* függvény meghívása megfelelő lehetne, azonban ezt a fejlesztőnek lehetősége van tetszőlegesen felüldefiniálni egy saját hash kód számító eljárással. A *System.Runtime.CompilerServices.RuntimeHelpers* osztály *GetHashCode* függvényét használjuk, amely kikerüli a felüldefiniált függvényt és az eredeti hash értéket adja vissza.

6.5 Teszteredmények

A kutatásom során elért eredmények működését és ezzel párhuzamosan teljesítményét is tesztelni kívántam. Erre a feladatra összesen négy különböző komplexitású alkalmazást választottam. Az alkalmazások kiválasztásában szerepet játszott az, hogy mennyi olyan műveletet tartalmaznak ezek a programok, amelyek nyomkövetésre kerülnek, és mennyi olyat tartalmaznak, amelyek valamilyen külső erőforrást használnak, amelyek elérésére várni kell. További fontos tudnivaló, hogy a BCL, azaz a .NET alapkönyvtáraiban végzett műveleteket nem kerülnek naplózásra, ugyanis a naplózó eljárás ezeket a modulokat explicit kihagyja, ezenfelül alapértelmezés szerint a PDB állományok

nem állnak rendelkezésre a BCL modulok esetében. Nem mindegy az, hogy mennyi .NET osztálykönyvtárbeli behívás történik és ezek mennyi ideig tartanak.

A választott négy alkalmazásból kettő alkalmazás nem illetve kevés és rövid lefutású BCL-hívást intéz. A harmadik alkalmazás több, de rövidebb hívást, míg a negyedik hosszú és sok behívást intéz.

Az alkalmazások konkrét karakterisztikája és működése a következő:

1. A *Counter* (számláló) nevű program természetes számok összegét számítja ki 1-től 100 000-ig brute force megoldással, azaz nem használ összegképletet, hanem elvégez minden elemi összeadást, méghozzá egy dedikált függvény meghívásával. Minden egyes részeredmény kiszámításakor egy „**.**” karaktert ír ki a képernyőre. A program futása során a kiíró művelet viszi el a legtöbb időt. Mivel a naplózó eljárás teljesítményét kívánom mérni, ezért a programot minimalizált állapotban futtatom.
2. Az *ITextSharp* [96] egy nyílt forráskódú könyvtár, amely többek között PDF dokumentumok olvasását és írását teszi könnyebbé. A tesztek során ennek a könyvtárnak a segítségével PDF állományokat állítottam elő. A könyvtár számos apró szöveg összefűzési művelettel rakja össze a PDF dokumentumot, amely szintén sok natív IL változókezelési hívást jelent, amit mind-mind naplózunk.
3. A *DiskReporter* nevű kis program rekurzívan bejárja a könyvtárfát a C meghajtó gyökerétől kezdve, majd az eredményekből egy XML kimutatást készít. A tesztjeim során 3245 könyvtár és 12849 fájlt érintettünk. Ez a program már több, azonban rövid osztálykönyvtár hívást használ, miközben kevesebb naplózandó IL szintű művelettel rendelkezik.
4. A *Mohican* egy C#-ban készült nyílt forráskódú, többszálú HTTP szerver, amelyet még 2004-ben készítettem el. A tesztek során a Mohican egy olyan 1.3 MB-os weboldalt szolgált ki, amely 20 olyan különböző képre hivatkozott legalább 10-10-szer. A web szerver sok és hosszú osztálykönyvtár hívást tartalmaz (leginkább hálózati és lemezműveletek)

A következő táblázat azt mutatja be, hogy a fent részletezett programok hány C# kódsorból állnak, valamint a naplózó eljárásom futtatása során hány naplóbejegyzés kerül a naplóba.

Alkalmazás megnevezése	C# kódsorok száma	Naplóbejegyzések száma
Counter	354	1 700 014
ITextSharp	202 249	5 364 020
DiskReporter	163	850 345
Mohican	3 354	97 353

6. táblázat: Az alkalmazások karakterisztikája

A pár szár soros programoktól kezdve, a pár ezer soros programon át több százezer soros programokat is vizsgáltam. A program kódsorainak száma természetesen nem befolyásolja a napló teljesítményét.

A következő táblázatban először a programok normál, napló nélküli futásának idejét szerepeltetem. A harmadik oszlop tartalmazza azt az esetet, amikor teljes funkcionalitású naplózás üzemel. A negyedik oszlopban a naplózó eljárások érdemi részét, azaz a kimeneti adatok formattálását valamint a kiírási műveletet kikommenteztem. Tehát ebben az esetben azt mérem, hogy mennyi idő szükséges a naplózó metódusok IL kódban való elhelyezéséhez, valamint az üres naplózó futás közbeni eljárások futtatásához.

Alkalmazás megnevezése	Normál futás ideje	Naplózás közbeni futás ideje	Üres naplózás futási ideje
Counter	00:03.18	00:09.98	00:03:43
ITextSharp	00:00.72	01:04.67	00:47.26
DiskReporter	00:16.61	00:32.74	00:17.02
Mohican	00:00.52	00:02.94	00:01.23

7. táblázat: Teljesítménymérés

A táblázatban szereplő időtartamok *perc:másodperc.századmásodperc* formátumúak.

A következő táblázatban azt szemléltetem, hogy a részletes napló létrehozása során milyen százalékos teljesítményt nyújtanak a különböző alkalmazások:

Alkalmazás megnevezése	Utasítás/másodperc naplózás nélkül	Utasítás/másodperc naplózással	%-os teljesítmény
Counter	534 595	170 372	31,86 %
ITextSharp	7 450 027	82 944	1,11 %
DiskReporter	51 194	25 972	50,73 %
Mohican	187 217	33 131	17,70 %

8. táblázat: Teljesítményelemzés (valódi napló)

A következő táblázatban azt mutatom be, hogy az üres naplózó eljárások futtatása során milyen százalékos teljesítményt nyújtanak a különböző alkalmazások:

Alkalmazás megnevezése	Utasítás/másodperc naplózás nélkül	Utasítás/másodperc üres naplózással	%-os teljesítmény
Counter	534 595	495 630	92,71 %
ITextSharp	7 450 027	113 500	1,52 %
DiskReporter	51 194	49 961	97,59 %
Mohican	187 217	79 148	42.28 %

9. táblázat: Teljesítményelemzés (üres napló)

Levonható a következtetés, hogy a sok, rövid alapvető műveletet (amelyeket naplózni is tudunk) alkalmazó programok sokkal nagyobb teljesítménycsökkentést szenvednek el, mint a sok, sőt hosszú osztálykönyvtár hívást intéző programok. A sok rövid műveletet tartalmazó alkalmazások esetében az üres naplózó eljárások is szignifikáns lassulást okoznak. Abban az esetben, ha hosszabb időtartamú műveleteket futtat a program, az üres naplózó eljárások behelyezése is kisebb hatást gyakorol a program futási teljesítményére.

További méréseket végeztem arra vonatkozóan, hogy a kiírandó szöveg formattálása vagy pedig az IO műveletek igényelnek-e több időt. Az a következtetés vonható le, hogy ugyan a formattálás is szignifikáns, de ennek ideje a napló lemezre írásához képest jelentéktelen.

További vizsgálatok kimutatták, hogy fix szélességű bináris formátumú naplóbejegyzések írása többszörös teljesítménynövekedést szolgáltathat, ugyanis ebben az esetben nem szükséges szövegformattálás valamint a kimenet olyan

formátumban van, amely egy az egyben írható a kimenetre (nem szükséges a .NET keretrendszer belső string->bináris konverziójának futtatása).

Mivel a napló elemzését támogató alkalmazások [2] a szöveges formátumot preferálják ezért a bináris naplót továbbfejlesztési lehetőségnek tekintem.

A következő részletet a Mohican futása során keletkezett naplóból emeltem ki:

```
3FU11429296R67108918:System.String:
3PU9040679R1:System.String:HTTP/1.1 200 OK
3FD19473824R67108918:System.String:HTTP/1.1 200 OK
3T133:4-133:48
3FU19473824R67108918:System.String:HTTP/1.1 200 OK
3LDV1:System.Boolean:True
3T136:4-136:16
3LDV0:System.Boolean:True
3T69L137:3-137:4
3LUV0:System.Boolean:True
3T45:5-45:63
3T78E425:3-425:4
3T426:4-426:30
3LD62619566R0:System.String:text/plain
3T429:4-429:5
3T430:5-430:41
3PU61646925R1:System.String:C:\Source\Mohican\wwwroot/index.html
3LDV1:System.Int32:37
3T432:5-432:50
3PU61646925R1:System.String:C:\Source\Mohican\wwwroot/index.html
3LUV1:System.Int32:37
```

39. ábra: Naplófájl részlet

6.6 Megjegyzések többszálú alkalmazásokra vonatkozóan

Többszálú alkalmazások készítése során a két legfontosabb probléma a következő:

1. Több szál által igénybe vett, osztott erőforrásokat használó kódblokkok szinkronizációja.
2. Időzítésből adódó un. versenyhelyzetek (race condition) kezelése.

Amikor két vagy több szál ugyanazt az osztott erőforrást használja, akkor el kell kerülnünk azt, hogy egy időben több mint egy szál műveletet végezhessen ezzel az erőforrással, azaz szinkronizálnunk kell a konkurens kódblokkokat, illetve kritikus szakaszokat kell kialakítanunk.

A versenyhelyzetek olyan esetekben alakulhatnak ki, amikor több szál ugyanazon az erőforráson kell, hogy műveletet végezzen, azonban nem mindegy ennek a sorrendje, időzítése. Ha ez a sorrendiség megsérül, akkor a program hibásan működhet, vagy biztonsági hiba válhat kihasználhatóvá.

Ha az eredeti programkódba naplózó eljárást injektálunk, mint ahogy azt az általam készített megoldás is teszi, akkor módosulhatnak az eredeti időzítési feltételezések, versenyhelyzet alakulhat ki a többszálú programok esetében.

Fontos megjegyezni, hogy egy versenyhelyzet előre nem várt hardver- és szoftverkörnyezetben a kód módosítása nélkül is kialakulhat.

A naplózó eljárások törzse tartalmaz egy .NET *lock*-ot, amely szinkronizált kódblokkot jelez a naplózó eljárások futtatásakor. Ez a megoldás garantálja azt, hogy a naplózó eljárások által használt osztott erőforrások (pl. napló kimenete) nem kerülhetnek egyidejűleg használat alá.

A szekvencia pont szintű naplózó eljárások az instrumentálás után csak statikusnak tekinthető adatokat alkalmazhatnak, ezért konkurens szál nem befolyásolhatja a naplóba kerülő adatokat.

A változó szintű naplózó eljárások során azonban felmerülhetnek problémák, ugyanis ott a naplózandó változó-hozzáférés esetében a változó értéke megváltozhat a naplózó eljárásnak való paraméterátadás, valamint a naplózás között eltelt idő alatt. Ha végiggondoljuk, ez az eset akkor és csak akkor adódhat, ha egy másik szál módosítja az osztott változó értékét. Ennek a változónak az olvasási és írási műveleteit akkor és csak akkor naplózza a megoldásom, ha ahhoz hozzáférés történt. Többszálú programok esetében azonban követelmény, hogy az osztott változókhoz történő hozzáférést megfelelően szinkronizált környezetben kell végrehajtani. Az ilyen környezetek tulajdonsága az, hogy a fent említett osztott változókhoz történő hozzáférési anomália nem történhet meg. Ezért az mondható, hogy a változó értéke a naplózó eljárás felparaméterezése, valamint a tényleges naplózás között eltelt idő alatt akkor és csak akkor változhat meg, ha az eredeti, később instrumentálásra kerülő program fejlesztője nem tartotta be a többszálú programokra vonatkozó fejlesztési alapelveket.

Ami a többszálú programvégrehajtás természetéből adódóan nem garantálható az az, hogy közvetlenül szálváltások előtt, illetve utána bekövetkező események a tényleges történés sorrendjében kerülnek naplózásra. Ha egy esemény bekövetkezett, majd pedig egy másik szálra kerül a vezérlés a naplózó eljárás lefutása nélkül, akkor a szálra történő vezérlés-visszaadás után kerül csak a naplóbejegyzés a naplóba.

6.7 Az elért eredmények összegzése

A 1. fejezetben ismertettem azokat a tudományos illetve tudományos alapon kifejlődött gyakorlatban is alkalmazható módszereket, amelyek elősegítik a minőségi szoftverek elkészítését. A 2. fejezetben a .NET kertrendszer [14]

biztonsági és programminőségi szolgáltatásait vettem górcső alá. Megvizsgáltam azokat a .NET-alapú programfejlesztés során felhasználható eszközöket, amelyek segítséget nyújthatnak az 1. fejezetben azonosított módszerek alkalmazásánál.

Ebből a körültekintő elemzésből vált nyilvánvalóvá, hogy részletes futás idejű naplózó eljárás nem áll rendelkezésünkre.

A 6. fejezetben ezért célul tűztem ki, hogy azonosítom egy olyan futás idejű naplókészítő eljárással kapcsolatos követelményeket, amelyek akár még a dinamikus szeletelés esetében is helytállnak.

Ezek után a Debugger [1] segítségével próbáltam meg egy az elvárásoknak eleget tevő rendszert megtervezni és megalkotni. Mivel a Debuggerel támogatott módszer segítségével nem lehetséges a követelmények kielégítése, ezért a Profiler alapú megoldásra tértem át.

A Profiler alapú megoldás [9] minden igénynek megfelel. A megoldás megtervezése és kivitelezése során iteratív módszert választottam. Először a szekvencia pontok által határolt utasítások szintjén készítettem naplót. Miután ez a megoldás beváltotta a hozzá fűzött reményeket, továbbléptem a változó szintű napló létrehozásának irányába. Ebben a második fázisban meghatározható az, hogy melyik utasításban, milyen változók olvasására illetve írására kerül sor a program futása során. A módszer abban is különbözik a többi megközelítéstől [33], hogy non-intrusive, azaz nem igényli sem manuálisan, sem automatikus eszköz segítségével a program eredeti forráskódjának módosítását ahhoz, hogy részletes naplót tudjon generálni a futó programok esetében.

A módszert több, különböző futási karakterisztikával rendelkező alkalmazás esetében is teszteltem. Ennek tükrében megállapítottam, hogy a rendszer teljesítménye megfelelő, azonban még felülvizsgálatot igényel.

Fontos megemlíteni, hogy mivel a .NET egy nyelv-független rendszer ezért a C#, a Visual Basic, a Managed C++, vagy éppen a funkcionális F# nyelv felhasználóinak is előnyére válik, számukra is hasznos az általam kifejlesztett naplózó metodológia.

3. Tézis. Megmutattam egy részletes, futási idejű naplózó eljárás szükségességét a szabványos .NET platform felett. Definiáltam az eljárással kapcsolatos követelményeket, valamint egy olyan programozási nyelv független megvalósítást hoztam létre, amely nem igényli az eredeti forráskód módosítását. A megoldás párhuzamos környezetben is megfelelő minőséggel és teljesítménnyel üzemel.

A témával kapcsolatos kutatások eredményét [1][2][9] alatt ismertettem.

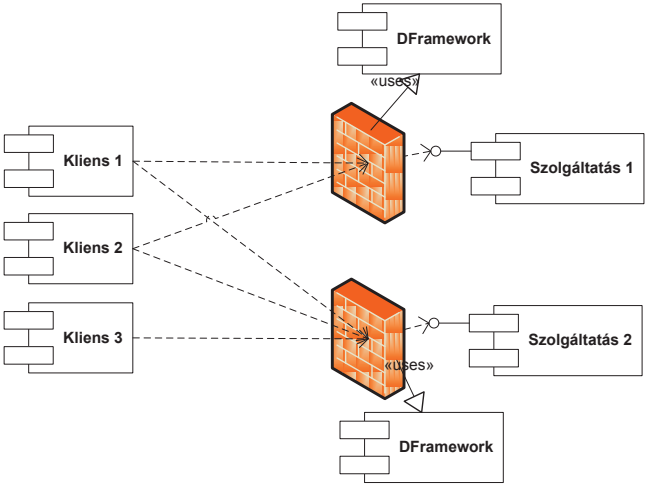
7 Integrált megoldások

Amikor egy kutatási eredmény illetve szoftvertermék elkészül, akkor fontos, hogy a már meglévő megoldásokkal könnyen integrálható legyen, egyszerűen tovább lehessen fejleszteni, illetve a későbbi megoldásokkal összekapcsolható, azokra kiterjeszthető legyen.

Ebben a fejezetben a II. részben elért eredmények egy kutatás alatt álló további bővítéséről, kiterjesztéséről szólok, valamint egy olyan koncepcióról, amely a II. részben és a 6. fejezetben leírtakat integrálja.

7.1 Az elosztott keretrendszer kiterjesztése

Az 5.5. fejezetben megmutattam a kész rendszer komponens szintű architektúráját. Egy több szolgáltatást tartalmazó komplex rendszer architektúráját a következő ábra szemlélteti:



40. ábra: DFramework rendszerarchitektúra

Az ábra szerint a három különböző kliens két egymástól független szolgáltatást használ. Mindegyik szolgáltatás előtt található egy tűzfallal jelzett komponens, amely az elosztott biztonsági keretrendszeremet jelképezi.

A „*Szolgáltatás n*” komponens a 4.5.2. fejezet első példája esetében magát a Hivatali Kapu szolgáltatást szimbolizálja, a „*Kliens n*” pedig a hozzá kapcsolódó hivatalokat. A második példa esetében az elfogadó munkafolyamatok, mint

szolgáltatások, a munkafolyamat résztvevői, mint kliensek értelmezhetők.

Az ábrából észrevehető, hogy a két szolgáltatás egymástól teljesen független, azaz az egyik szolgáltatás állapota alapján nem lehetséges megszorításokat tenni a másik metódusainak elérhetőségére, meghívhatóságára. Célom az, hogy ennek lehetőségét megteremtsem.

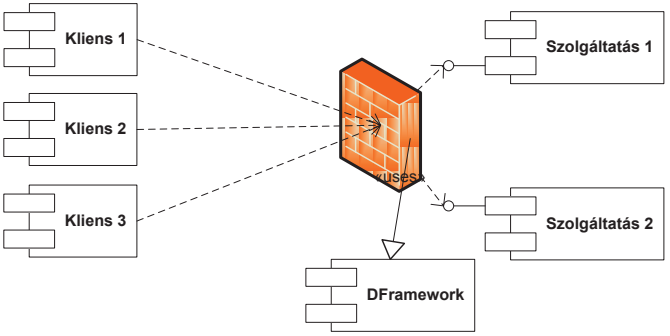
Visszatekintésképpen, a következő megszorítási kategóriákat vezettem be 4-ben:

1. A szofisztikált hozzáférés-vezérlési mechanizmusok elosztott alkalmazásokra való kiterjesztése – a hívó típusának vizsgálata
2. Az üzleti szolgáltatások és a munkafolyamatok integrációja
3. A hívó identitásának, valamint különböző identitás alapú szabályoknak az alkalmazása
4. Hálózati megszorítások részletezettségének javítása

A fentiek közül a második illetve a harmadik pont az, amely a szolgáltatások belső állapotának ismeretét igénylik. A második esetben a munkafolyamat aktuális állapotát kell ismernünk, a harmadik esetben pedig a szabályok kiértékeléséhez a szolgáltatás metódusok hívási történetét kell tudnunk.

Amennyiben több szolgáltatásra egy közös megszorításcsomagot akarunk értelmezni, akkor a kontextusfüggő állapotaikat egy közös tárban kell nyilvántartani.

A fent vázolt problémát a következő architektúra oldja meg:



41. ábra: DFramework továbbfejlesztett rendszerarchitektúra

Amennyiben ezt az architektúrát alkalmazzuk, akkor lehetőség van olyan formális szabályok definiálására, amely a különböző szolgáltatások mögöttes munkafolyamatainak állapotát illetve metódusainak hívási történetét is figyelembe

veszi. Például a következő feltételeket is megfogalmazhatnánk:

1. Az “A” szolgáltatás “a” metódusa akkor hívható meg, ha a “B” szolgáltatás “x” állapotban van
2. Az “A” szolgáltatás “a” metódusát akkor hívhatja meg az “u” felhasználó, ha a “B” szolgáltatás “b” metódusát is “u” hívta meg

Az első esetre a példa az, hogy egy pénzügyi ellenjegyzési munkafolyamat csak akkor indítható el, ha egy többlépcsős elfogadási munkafolyamat már legalább az osztályvezetői szinten tartózkodik. A második esetet prezentálhatja, amikor egy számla kiállítását végző műveletet ugyanannak a felhasználónak kell meghívnia, mint aki a számla alapját képező teljesítési igazolást is rögzítette.

Természetesen ezek a kiterjesztések azt igénylik, hogy a szolgáltatások egy közös munkamenet (session) azonosítóval dolgozzanak. A munkamenet azonosító felelős azért, hogy segítségével a különböző szolgáltatások összekapcsolhatók legyenek, a különböző, de összefüggő szolgáltatáspéldányok egy állapotinformáció tárat használhassanak. A munkamenet azonosító segítségével ez az állapotinformáció tár bármikor elérhető a munkamenethez tartozó szolgáltatás példányokból.

Megjegyzem, hogy a munkamenetek használata ellentmond annak, hogy a SOA szolgáltatásainak állapotmentesen célszerű üzemelniük.

7.2 A futás idejű napló további alkalmazásai

Ebben a fejezetben legelőször felelevenítem, hogy milyen használati esetek céljából készítettem el a korábban felvázolt naplózó eljárást, majd pedig új használati eseteket mutatok be.

A 6. fejezetben egy olyan naplózó eljárást mutattam be, amely

3. támogatja a .NET alapú alkalmazások futás közbeni programútjának és változó-hozzáféréseinek rögzítését későbbi vizsgálat céljából és ezáltal
4. felhasználható egy dinamikus szeletelő algoritmus bemeneteként is.

Amikor egy programot fejlesztünk, akkor az idő nagy részét debuggolással kell tölteni. A komplexebb alkalmazások debuggolása nehéz feladat, mivel nagy mennyiségű programkódot szükséges átlátni, valamint a debugger eszköz futtatása során olyan mellékhatásokkal szembesülhetünk (többszörös kiértékelés, threading problémák, stb.), amelyek tovább nehezítik a munkát.

Amennyiben a program futásáról egy naplót készítünk, akkor ezek a mellékhatások nem, vagy csak csökkentett mértékben jelentkeznek még akkor is, ha az eredeti működésre ortogonális művelet a naplózás és a változók értékeit nem

módosítja. Nyilvánvalóan nem tud teljesen mellékhatás-mentes lenni egy naplózó megoldás, ugyanis a naplóbejegyzések lefuttatása időbe kerül, amely időzítési kérdéseket vet fel.

Ha dinamikus szeletelést végzünk, akkor az átlátni szükséges programkód utasításinak száma drasztikusan csökken, ezzel enyhítve azt a mentális megterhelést, amely a programozóra hárul a hibakeresési feladat ellátása során.

A dinamikus szeletelés bemenete egy olyan napló [71], amely azokat az utasításokat tartalmazza, amelyek végrehajtásra kerültek a program futása során, tehát a klasszikus koncepció szerint nem szükséges a változók felhasználását naplózni, ugyanis a definiált illetve felhasznált változókat a forráskód elemzése segítségével határozzák meg ezek az algoritmusok [1].

Az általam adott naplózó megoldás részletességéből adódóan a komplexebb kifejezéseket tartalmazó .NET programok magas részletességű dinamikus szeletelését is lehetővé teszi.

A következő ábrán egy egyszerű programot fogok bemutatni. A szeletelési kritérium a következő lesz: (<>, *ut1*, *x*). Kiemeltem azokat az utasításokat, amelyek futtatásra kerültek az *ut1* utasításig:

```
class Program
{
    static int CalcA() { return 1; }
    static int CalcB() { return 1; }
    static int CalcC() { return 2; }
    static int CalcD() { return 3; }

    static void Main(string[] args)
    {
        int a = CalcA();
        int b = CalcB();
        int c = CalcC();
        int d = CalcD();
        int x = 0;

        if (a == b || c == d)
        {
            x = a + b;
        }
        else
        {
            x = b + d;
        }

        ut1: Console.WriteLine(x);
    }
}
```

42. ábra: Naplózott programlépések (dinamikus szelet)

Ha végiggondoljuk, akkor amennyiben utasítás szintű naplózásról beszélünk, akkor a dinamikus programszelet megegyezik a kiemelt utasításokkal (az *int x* =

0; utasítás kivételével), ugyanis nem ismert, hogy az *a == b* vagy pedig a *c == d* feltétel teljesülése miatt vált-e igazzá az *if* feltétele. A .NET pedig pont így kezeli a szekvencia pont határok között álló utasításokat, azaz egy logikai kifejezés minden komponense ugyanazon komplex utasításhoz tartozik, ez kerül a naplóba is.

Amennyiben azt is naplózzuk, hogy milyen változók kerültek olvasásra, illetve definiálásra, akkor kiderül, hogy az if utasításban csak és kizárólag az *a* és a *b* változót olvasta a program, a *c* és a *d* változót már nem az *if* lusta kiértékelése miatt.

Ebből következően a keletkezett programszelet a következő lesz:

```
class Program
{
    static int CalcA() { return 1; }
    static int CalcB() { return 1; }
    static int CalcC() { return 2; }
    static int CalcD() { return 3; }

    static void Main(string[] args)
    {
        int a = CalcA();
        int b = CalcB();
        int c = CalcC();
        int d = CalcD();
        int x = 0;

        if (a == b || c == d)
        {
            x = a + b;
        }
        else
        {
            x = b + d;
        }

        ut1: Console.WriteLine(x);
    }
}
```

43. ábra: Pontosabb dinamikus szelet

Több megoldás [20] [76] látott már napvilágot, amelyek az imént említett részletességre képesek, azonban a .NET környezetre egyik sem adaptálható könnyen. Sok implementáció C, C++ vagy éppen a Java nyelvre specializálódott. Ezek közül több intruzív módon működik, azaz az eredeti forráskódot módosítja, amelyhez a forrásnyelvi szöveg elemzése és átalakítása szükséges. A C# nyelv és a .NET környezet sokban eltér ezektől az adatok, változók kezelése valamint a programnyelvi szolgáltatások szempontjából is.

További felhasználási mód a napló alapján történő ellenőrzése annak, hogy a program megfelel-e a specifikációnak. Ehhez az szükséges, hogy a változók

értékeit is naplózzuk. Egyszerű változók esetében a napló kimenete a változók aktuális értékeit is tartalmazza.

Amennyiben a programunkat előfeltételekkel, utófeltételekkel, illetve invariánsokkal látjuk el ily módon specifikációt adunk meg, akkor ennek segítségével ellenőrizhetjük a program helyességét.

Két megoldási mód kínálkozik erre:

1. A naplózott bejegyzéseket futás időben értékeljük ki, és vetjük össze a specifikációval
2. A program teljes lefutása után bejárt program-utat, illetve változó értékeket validáljuk a specifikáció segítségével

Természetesen mindkét megközelítésnek vannak előnyei és a hátrányai is. Az első esetben kisebb tárhely szükséges, ugyanis a már nem hatásos naplóbejegyzések eldobhatók, azonban bonyolultabb, nagyobb számítási kapacitású algoritmusról beszélhetünk. Kifejezett előny, hogy ebben az esetben azonnal értesülünk a specifikáció megsértéséről. A második esetben a teljes napló eltárolása szükséges, azonban egyszerűbb algoritmus készíthető, amivel csak a teljes lefutás után kapjuk meg az eredményt.

További lehetőségek is nyílnak a napló felhasználására. A II. fejezetben ismertetett elosztott algoritmus működésének nem elosztott rendszerekre történő adaptálásával vagy a napló elosztott alkalmazásokra való kiterjesztésével ez a két módszer akár össze is kapcsolható. Röviden összefoglalva a II. fejezetben felvázolt megszorítások a fent említett két módszer valamelyikével validálhatók. A különbség annyi, hogy a specifikációs feltételek alatt a metódusok elérésére jogosult felhasználókat, programkódokat értem.

IV. Összefoglalás

8 Összegzés

Dolgozatomat a minőség és a biztonság általános fogalmának bemutatásával kezdtem. Ezt követően ezen definíciók túl széles, tág jellegéből adódóan olyan szűkebb részhalmazokat vettem alapul, amelyek már könnyen kezelhetők, valamint a szoftverfejlesztés fázisaira koncentrálnak. Röviden ismertettem azt a feltörekvő platformot, a .NET-et, amelyhez kutatásom során plusz értéket adtam, valamint új eredmények előállításának támogatására használtam. Ezen felül beavattam az olvasót az elosztott alkalmazások azon gyakorlati fejlesztési, architekturális, illetve módszertani alapjaiba, amelyek egyre nagyobb teret hódítanak az ipari felhasználók körében is. Ezek a bevezető jellegű fejezetek az I. részben kerültek bemutatásra.

A dolgozatban felvázolt minőséget és biztonságot támogató módszereket azon fejlesztési fázisok sorrendjében tárgyaltam, mint ahogy azok egymás után következnek. Először a tervezési, majd pedig a fejlesztési-tesztelési szakaszhoz adtam hozzá újszerű módszereket.

A II. rész tehát már egy saját eredményt mutat be, amellyel a biztonságos elosztott alkalmazások tervezését kívánom elősegíteni.

A 4. fejezetben elsőként azokat a problémákat vázoltam fel, amelyek nemcsak számomra, hanem több kutatónak is szemet szúrt. Megvizsgáltam a kapcsolódó munkákat, amelyekben hiányosságokat fedeztem fel. Az eddig egymástól függetlenül létező, nehézkesen integrálható üzleti szolgáltatásokat, munkafolyamatokat, hozzáférés-vezérlést, szabály alapú jogosultságkezelést kötöttem össze. A klasszikus hozzáférés-vezérlés egy kiterjesztését definiáltam az elosztott alkalmazások kontextusába. Mivel az elosztott alkalmazások platform független rendszerként kell, hogy üzemeljenek, ezért először tisztán formális eszközök segítségével definiáltam a biztonsági szolgáltatások működését. Formálisan definiáltam a biztonsági megszorítások körét, valamint a legális metódushívás definícióját is. A formalizmus használhatóságát ipari esettanulmányokon is szemléltettem.

A 5. fejezetben a formális modell alapján egy egyértelmű implementációt készítettem a .NET platform felhasználásával. Egzaktul meghatároztam a rendszer futási környezetét, a komponensek szerepköreit, a felhasznált paradigmákat, valamint ezek szerepét. A formális modell megszorításaira egy átírási módszert dolgoztam ki. Törekedtem arra, hogy olyan interfészt adjak a termékként létrejött keretrendszerhez, amely egyszerűsége révén könnyen használható, automatizálható, és hatékony segítséggé válhat a használói számára. Az eredményt a korábban felvázolt esettanulmányokon is teszteltem.

A III. fejezetben a fejlesztési-tervezési fázisra tértem át. A dinamikus programszeletelés algoritmusának egyik bemenete egy részletes napló, amely egyaránt hasznos a futási környezetben, a futás alatt felmerülő problémák detektálásában, elemzésében, majd pedig megoldásában. A .NET keretrendszert egy olyan naplózó eljárással vérteztem fel, amely ezeknek a feltételeknek képes eleget tenni. Fontos megemlíteni, hogy mivel a .NET egy nyelv-független rendszer ezért a Standard C#, a Visual Basic, az Managed C++, vagy éppen a funkcionális F# nyelv felhasználóinak is előnyére válik ez a napló.

A naplózó eljárással kapcsolatos előkészítési, kutatási, ismeretgyűjtési munkát, a saját eredményeket leíró 6. fejezetben tárgyaltam. Ugyanitt kapott helyet a technológiai megvalósítás leírása, a teszteredmények bemutatása, valamint az iteratív kutatási módszer ismertetése is.

A megoldásom a nyelvfüggetlenség mellett abban is egyedülálló, hogy nem igényli az eredeti program forráskódjának módosítását.

Szintén a III. részben található 7. fejezetben olyan integrált megoldásokat vázoltam fel, amelyek a dolgozatban bemutatott eredmények felhasználási módjait, összekapcsolását ismertetik, szorgalmazzák.

Megmutattam, hogy milyen architekturális módosítások szükségesek ahhoz, hogy komplexebb megszorításokat is ki lehessen fejezni az elosztott alkalmazások biztonságosabbá tétele céljából.

Ismertettem, hogy milyen felhasználási esetei vannak a futás idejű naplónak, valamint rámutattam arra is, hogy az általam kidolgozott eljárás hogyan teszi lehetővé pontosabb dinamikus szeletelési megoldások kifejlesztését. Szintén fontos szerepet kaphat a napló a program helyességvizsgálatánál, valamint a két tárgyalt módszer integrálása során létrejött napló alapú jogosultságellenőrzésnél.

Bízom benne, hogy a jövőben ezek az eredmények hozzájárulnak majd a minőségi programok előállításához, valamint az informatikai biztonság tudatosabb használatához.

8.1 A dolgozat eredményei

1. Tézis. Megmutattam a jelenlegi elosztott alkalmazások esetében használatos hozzáférés-vezérlő mechanizmusokban található korlátokat. Egy olyan formális modellt definiáltam, amely megválaszolja az elosztott hozzáférés-vezérlés legfontosabb kérdéseit, összekapcsolja a szolgáltatások és munkafolyamatok jogosultságkezelését, kiterjeszti a szerepkör alapú valamint a hálózati szegmensekhez tartozó jogosultságkezelést, valamint absztraktságából adódóan garantálja a platform- és implementációfüggetlenséget. A formális modell alkalmazhatóságát több ipari esettanulmányon keresztül is validáltam.

A témával kapcsolatos eredményeket [4] [7][8] alatt publikáltam.

2. Tézis. A formális modellt felhasználva egy keretrendszert terveztem, amely alapján működő implementációt készítettem szabványos C# nyelven a .NET platformra. Megmutattam egy átírási módszert, amellyel a formális modell által definiált megszorítások egyértelműen átírhatók C# nyelvre. A keretrendszer segítségével megvalósítottam a formálisan is definiált esettanulmányokat.

A témával kapcsolatos eredményeket [8] alatt vázoltam fel.

3. Tézis. Megmutattam egy részletes, futási idejű naplózó eljárás szükségességét a szabványos .NET platform felett. Definiáltam az eljárással kapcsolatos követelményeket, valamint egy olyan programozási nyelv független megvalósítást hoztam létre, amely nem igényli az eredeti forráskód módosítását. A megoldás párhuzamos környezetben is megfelelő minőséggel és teljesítménnyel üzemel.

A témával kapcsolatos kutatások eredményét [1][2][9] alatt ismertettem.

8.2 Angol nyelvű összefoglalás

Summary

In this theses work, after discussing the concept of software quality and security, I showed the precise sub-area of these that concentrates on the phases of software development. I discussed, in brief, the main concepts of the emerging standard .NET platform, where further improvements have been made via my research work. After this I foreshown the most modern development, the architectural and methodological foundations of practical distributed system creation, that are nowadays growing up in the industry.

The outlined quality and security supporting methodologies were discussed in the order they are employed in an industrial project; I at first added on novel methodologies to the design phase, and then have covered the development and testing phases.

I have been analyzing a problem that caught not only my eye but those of many other researchers. The topic is the relationship between business services, workflows, visibility and rule-based access control mechanisms of distributed applications. My contribution has been to combine, improve and step up the granularity of the distributed system security. Since a distributed application must operate platform-independently, a formal model of the functions of my security services is first of all defined. Specifically I have characterized the formal restrictions and the concept of the legal method call. The usability of my model is also demonstrated through industrial case studies.

After this, I created an exact implementation based on the .NET platform, where a rewriting system is used to make the work of the developer simpler and more automatable. The operation is demonstrated via the industrial case studies, introduced before.

In the development and testing phase it is essential to maximally support software testing and the detecting of errors. A detailed runtime trace of the program execution can help in this serious work; moreover, the input of the dynamic program slicing algorithms is also a detailed runtime trace. Based on the .NET technology I have built up a language independent tracing mechanism to support the above expectations. The way of working is non-intrusive, so it does not require the original source code of the program is altered.

Finally, I further clarified the applicability of the methods, and have shown a way to integrate the two way of working to support trace-based security checks.

To support more complex application scenarios some minor changes to the architecture of the distributed framework are required, and these have been shown in my study. Furthermore, it is also shown why my way of operating is able to support more precise dynamic slicing algorithms.

I hope that these ways of operating will be able to support the creation of higher quality software and more secure application development.

Hivatkozások

1. Pócza, K., Biczó, M., Porkoláb, Z.: Cross-language Program Slicing in the.NET Framework. In : Conference proceedings of.NET Technologies 2005, Plzen (Czech Republic), pp.141-150 (2005)

2. Pócza, K., Biczó, M., Porkoláb, Z.: Towards Effective Runtime Trace Generation Techniques in the.NET Framework. In : Short communication papers proceedings of.NET Technologies 2006, Plzen (Czech Republic), pp.9-16 (2006)

3. Biczó, M., Pócza, K., Porkoláb, Z.: A Cache-Based Interprocedural Static Slicing Algorithm. In : Proceedings of the 7th International Conference on Applied Informatics (ICAI), Eger (Hungary), pp.207-218 (2007)

4. Pócza, K., Biczó, M., Porkoláb, Z.: Runtime Access Control in C#. In : Proceedings of the 7th International Conference on Applied Informatics (ICAI), Eger (Hungary), pp.28-31 (2007)

5. Pócza, K., Pataki, N.: An Improvement on the Access Control Features of C#. In : Proceedings of Sixteenth Electrotechnical and Computer Science Conference (ERK 2007), Portoroz, vol. B, pp.33-41 (2007)

6. Biczó, M., Pócza, K., Forgács, I., Porkoláb, Z.: A New Concept of Effective Regression Test Generation in a C++ Specific Environment. Acta Cybernetica 18(3), 481-501 (2008)

7. Biczó, M., Pócza, K., Porkoláb, Z.: Runtime access control in C# 3.0 using extension methods. Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica 30, 41-60 (2009)

8. Pócza, K., Biczó, M., Porkoláb, Z.: Securing Distributed.NET Applications Using Advanced Runtime Access Control. Frentiu et al ed.: Studia Universitatis Babes-Bolyai Informatica LIII(2008/2), 39-54 (2008)

9. Pócza, K., Biczó, M., Porkoláb, Z.: Towards detailed trace generation using the profiler in the.NET Framework. Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica 30, 21-40 (2009)

10. Pócza, K., Biczó, M., Porkoláb, Z.: docx2tex: Word 2007 to TeX. TUGBoat, TUG 2008 Conference Proceedings 29(3), 392-400 (2008)

11. Pócza, K., Biczó, M., Porkoláb, Z.: FC#: Designing an Internal Functional DSL to C# 3.0. In : Proceedings of the Implementation and Application of Functional Languages 20th International Symposium, IFL 2008, Hatfield, Hertfordshire (UK), vol. Technical Report no. 474 (2008), pp.299-314 (2008)

12. Biczó, M., Pócza, K.: Generating Functional Implementations of Finite State Automata in C# 3.0. Electronic Notes in Theoretical Computer Science (ENTCS) 238(2), 3-12 (2009)

13. Agrawal, H., Horgan, J.: Dynamic program slicing. Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, 246 - 256 (1990)

14. Albert, I., Balássy, G., Charaf, H., Erdélyi, T., Horváth, Á., Levendovszky, T., Péteri, S., Rajacsics, T.: A.NET Framework és programozása. Szak Kiadó (2004)

15. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)

16. Barros, A., Decker, G., Dumas, M., Weber, F.: Correlation Patterns in Service-Oriented Architectures. In : Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007), Springer Verlag, pp.245-259 (2007)

17. Bauer, L., Garriss, S., Reiter, M.: Efficient Proving for Practical Distributed Access-Control Systems. Computer Security - ESORICS 2007, LNCS, 19-37 (2007)

18. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change 2nd edn. Addison-Wesley

19. Bell, M.: "Introduction to Service-Oriented Modeling". Service-Oriented Modeling: Service Analysis, Design, and Architecture. Wiley & Sons (2008)

20. Beszédes, Á., Gergely, T., Gyimóthy, T.: Graph-Less Dynamic Dependence-Based Dynamic Slicing Algorithms. In : Proceedings of the 6th IEEE Int'l Workshop on Source Code Analysis and Manipulation, pp.21-30 (2006)

21. Beszédes, Á., Gergely, T., Szabó, Z., Csirik, J., Tibor, G.: Dynamic Slicing Method for Maintenance of Large C Programs. In : Proceedings of the 5th CSMR 2001, pp.105-113 (2001)

22. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The role of trust management in distributed systems security. In : Secure Internet Programming. Springer Verlag, pp.185-210 (1999)

23. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys 17(4), 471-522 (1985)

24. Cooney, D., Dumas, M., P., R.: GPSL: A Programming Language for Service Implementation. In : Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE), pp.3-17 (2006)

25. Crampton, J.: A reference monitor for workflow systems with constrained task execution. In : Proceedings of the 10th ACM Symposium on Access Control Models and Technologies, pp.38-47 (2005)

26. Csörnyei, Z.: Fordítóprogramok. Typotex (2006)

27. Czarnecki, K., Eisenecker, U., Glück, R., Vandevoorde, D., Veldhuizen, T.: Generative programming and active libraries. LNCS 1766, 25-39 (2000)

28. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)

29. Damianou, N., Dulay, N., Lupu, E., Sloman, M., Tonouchi, T.: Policy Tools for Domain Based Distributed Systems Management. In : IFIP/IEEE Symposium on Network Operations and Management (2002)

30. Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453 - 457 (1975)

31. Domingos, D., Silva, A., Veiga, P.: Workflow Access Control from a Business Perspective. In : International Conference on Enterprise Information Systems (2004)

32. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2003)

33. Ferenc, R., Beszédes, Á., Gyimóthy, T.: Extracting Facts with Columbus from C++ Code. In : Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), pp.4-8 (2004)

34. Fowler, M.: Refactoring. Improving the Design of Existing Code. Addison-Wesley (1999)

35. Frank, E.: Developing Distributed Enterprise Applications With the MS Common Object Model. Hungry Minds (1997)

36. Fóthi, Á.: Bevezetés a programozáshoz. ELTE Eötvös Kiadó (2007)

37. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

38. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: A Comparative Study of Language Support for Generic Programming. In : Proceedings of the 18th ACM SIGPLAN OOPSLA, pp.115-134 (2003)

39. Gönczy, L., Heckel, R., Varró, D.: Model-Based Testing of Service Infrastructure Components. Proc. Testing of Software and Communicating Systems, 19th IFIP

TC6/WG6.1 International Conference, LNCS 4581, 155-170 (2007)

40. Gronmo, R., Jaeger, M., Wombacher, A.: A Service Composition Construct to Support Iterative Development. In : Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering, pp.230-244 (2007)

41. Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576-580 (1969)

42. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley (2003)

43. Hunyadvári, L., Manhertz, T.: Automaták és formális nyelvek. Elektronikus előadásjegyzet. (http://aszt.inf.elte.hu/~hunlaci/book.pdf)

44. Juric, M., Mathew, B., Sarang, P.: Business Process Execution Language for Web Services: BPEL and BPEL4WS. Packt Publishing (2004)

45. Kaner, C., Falk, J., Nguyen, H.: Testing Computer Software 2nd edn. Wiley (1999)

46. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. Proceedings of ECOOP, Finland. Springer-Verlag LNCS 1241, 220–242 (1997)

47. Koopman, P., Plasmeijer, R., Eekelen, M., Smetsers, S.: Functional programming in Clean. (2002)

48. Kozsik, T., Csörnyei, Z., Horváth, Z., Király, R., Kitlei, R., Lövei, L., Nagy, T., Tóth, M., Víg, A.: Use cases for refactoring in Erlang. Lecture Notes in Computer Science 5161, 264-298 (2008)

49. Krinke, J.: Advanced Slicing of Sequential and Concurrent Programs, PhD Thesis. Universität Passau (2003)

50. Liberty, J., Xie, D.: Programming C# 3.0. O'Reilly, ISBN 0-596-00489-3 (2008)

51. Lövy, J.: Programming.NET Components. O'Reilly (2003)

52. Marshall, D.: Programming Microsoft Visual C# 2005: The Language. Microsoft Press (2006)

53. Meyer, B.: Applying "Design by Contract". Computer (IEEE) 25(10), 40-51 (1992)

54. Meyer, B.: Eiffel: The Language. Prentice Hall (1991)

55. Meyer, B.: Object-Oriented Software Construction 2nd edn. Prentice Hall (1997)

56. Mikunov, A.: Rewrite MSIL Code on the Fly with the.NET Framework Profiling API. MSDN magazine (September 2003)

57. Nyékyné Gaizler, J., ed.: Programozási nyelvek. Kiskapu Kft. (2003)

58. Ottenstein, K., Ottenstein, L.: The program dependence graph in software development environment. ACM SIGPLAN Notices 19(5), 177-184 (1984)

59. Pellegrino, M.: Improve Your Understanding of.NET Internals by Building a Debugger for Managed Code. MSDN Magazine (november 2002)

60. Porkoláb, Z., Mihalicza, J., Sipos, Á.: Debugging C++ Template Metaprograms. In : The ACM Digital Library (GPCE), pp.255-264 (2006)

61. Rising, L., Janoff, N. S.: The Scrum Software Development Process for Small Teams. (2000)

62. Rivest, R., Leiserson, C., Cormen, T.: Algoritmusok. Műszaki Könyvkiadó Kft. (2001)

63. Sike, S., Varga, L.: Szoftvertechnológia és UML. ELTE Eötvös Kiadó

64. Sipos, Á., Porkoláb, Z., Zsók, V.: Meta <Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms. Studia Universitatis Babes-Bolyai Informatica LIII(2), 55-66 (2008)

65. Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In : Proceedings of OOPSLA '86, pp.38-45 (1986)

66. Stroustrup, B.: The C++ Programming Language Special Edition. Addison-Wesley (2000)

67. Tari, Z., Bukhre, O.: Fundamentals of Distributed Object Systems: The CORBA Perspective. Wiley (2001)

68. Tejfel, M.: Funkcionális programozási nyelvek helyességvizsgálata. Phd disszertáció (2008)

69. Thompson, S.: Haskell: The Craft of Functional Programming 2nd edn. Addison-Wesley, ISBN 0-201-34275-8 (1999)

70. Thomsen, D., O'Brien, D., Bogle, J.: Role Based Access Control Framework for Network Enterprises. In : Proceedings of 14th Annual Computer Security Applications Conference, pp.50 - 58 (1998)

71. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3), 121-189 (1995)

72. Ullmann, J., Widom, J.: Adatbázisrendszerek - alapvetés második, átdolgozott kiadás edn. Panem (2008)

73. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.: Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall (2005)

74. Wei, X., Jun, W., Yu, L., Jing, L.: SOWAC: a Service-Oriented Workflow Access Control Model. In : Proceedings of the 28th Annual International Computer Security and Applications Conferences, pp.128-134 (2004)

75. Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering. SE-10(4), 325-357 (1984)

76. Zhang, X., Gupta, R., Zhang, Y.: Precise dynamic slicing algorithms. In : Proceedings of International Conference on Software Engineering, pp.319-329 (2003)

77. Elektronikuskormányzat-központ - Ajánlások: http://www.ekk.gov.hu/hu/kib/ajanlasok.

78. OASIS Standard - Web Services Business Process Execution Language Version 2.0: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

79. SOAP Specifications: http://www.w3.org/TR/soap/.

80. Standard ECMA-334 C# Language Specification: http://www.ecma-international.org/publications/standards/Ecma-334.htm.

81. Standard ECMA-335 Common Language Infrastructure (CLI): http://www.ecma-international.org/publications/standards/Ecma-335.htm.

82. Standard ISO 17799: http://www.symantec.com/region/hu/resources/ISO17799.html.

83. Standard ISO 9000:2005: http://www.iso.org/iso/catalogue_detail.htm?csnumber=42180.

84. Standard ISO/IEC 23270:2006 - C# Programming Language: http://www.iso.org/iso/catalogue_detail.htm?csnumber=42926.

85. Standard ISO/IEC 23271:2006 - Common Language Infrastructure: http://www.iso.org/iso/catalogue_detail.htm?csnumber=42927.

86. W3C Submission - Web Services Addressing (WS-Addressing): http://www.w3.org/Submission/ws-addressing/.

87. A.NET Framework és Programozása II tárgy honlapja: http://avalon.inf.elte.hu/edu/net2/default.aspx.

88. Agile Alliance: http://www.agilealliance.org/.

89. BSD licensz: http://en.wikipedia.org/wiki/BSD_licenses.

90. Biztostű: http://www.biztostu.hu/course/view.php?id=21.

91. Bruce Schneider: http://www.schneier.com/.

92. CodePlex: http://www.codeplex.com/.

93. David Broman's CLR Profiling API Blog: http://blogs.msdn.com/davbr/.

94. IBM WebSphere: http://www-01.ibm.com/software/hu/websphere/.

95. ISA Server: http://www.microsoft.com/hun/isaserver/default.mspx.

96. ITextSharp: http://itextsharp.sourceforge.net/.

97. Igazságügyi és Rendészeti Minisztérium - e-Beszámoló Rendszer: http://www.e-beszamolo.irm.hu/.

98. Java Developer Network: http://java.sun.com/.

99. Microsoft.NET Framework: http://www.microsoft.com/net/.

100. K2 Workflow: http://www.k2.com/en/index.aspx.

101. Mike Stall's.NET Debugging Blog: http://blogs.msdn.com/jmstall/.

102. Mono keretrendszer: http://www.mono-project.com/Main_Page.

103. MSF for Agile Software Development Process Guidance: http://www.microsoft.com/downloads/details.aspx?FamilyId=9F3EA426-C2B2-4264-BA0F-35A021D85234&displaylang=en.

104. Object-relational mapping: http://en.wikipedia.org/wiki/Object-relational_mapping.

105. Official Microsoft IIS Site: http://www.iis.net/.

106. Service-oriented architecture: http://en.wikipedia.org/wiki/Service-oriented_architecture.

107. Ügyfélkapu: http://www.magyarorszag.hu/.

108. Windows Communication Foundation: http://msdn.microsoft.com/en-us/netframework/aa663324.aspx.

109. Windows Workflow Foundation: http://msdn.microsoft.com/en-us/netframework/aa663328.aspx.

110. Yannis Smaragdakis: PhD Rants and Raves: http://www.cs.umass.edu/~yannis/phd-slides.pdf.